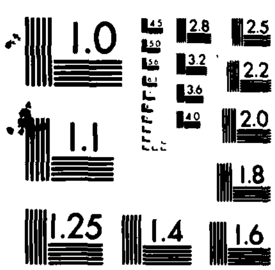
AD-A108 528 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME 10. DIRECTIVES.(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-10 NL

3

1 OF 3

AD-A 108528



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PHOTOGRAPH THIS SHEET

ADA108528

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Softech, Inc.
Waltham, MA
The JOVIAL (J73) Workbook
Nov. 81

DOCUMENT IDENTIFICATION

Contract F30602-79-C-0040 Rept.No. RADC-TR-81-333
Vols X–XII (of 15)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS GRA&I	☒
DTIC TAB	☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
DEC 14 1981
S D
D

DATE ACCESSIONED

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-333, Vols X-XII (of 15)
Interim Report
November 1981

THE JOVIAL (J73) WORKBOOK

SofTech, Inc.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-333, Vols I-XII (of 15) have been reviewed and are approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-333, Vols X - XII (of 15)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE JOVIAL (J73) WORKBOOK		5. TYPE OF REPORT & PERIOD COVERED Interim Report Dec 79 - Oct 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) N/A		8. CONTRACT OR GRANT NUMBER(s) F30602-79-C-0040
9. PERFORMING ORGANIZATION NAME AND ADDRESS SofTech, Inc. 460 Totten Pond Rd Waltham MA 02154		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 33126F 20220403
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE November 1981
		13. NUMBER OF PAGES 315
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas A. White (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL (J73)
MIL-STD-1589A
Video Course
Higher Order Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The JOVIAL (J73) Workbook is only one portion of a self-instructional JOVIAL (J73) training course. In addition to the programmed-learning primer/workbooks, are video taped lectures. The workbooks are formatted to consist of fifteen (15) segments bound in three (3) volumes covering each particular language capability. A video tape lecture was prepared for each workbook segment. This course is taught in two parts. Part I contains twelve (12) segments in Volumes I and II of the workbook; Part II

Volume III contains three (3) segments. There is a brief explanatory introduction at the beginning of the course. Each of the individual segments deals with a specific feature of the JOVIAL language. The video tapes act as an overview to outline particular points that are followed up in the written workbooks. Each tape runs a maximum of 25 minutes and contains an average of 15 graphic each.

THE JOVIAL (J73) WORKBOOK

VOLUME 10

DIRECTIVES

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 10 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

This workbook discusses compiler directives which provide supplemental information to the compiler for module linkage, optimization, register control, listing options and other miscellaneous functions. Each directive is discussed in detail along with its placement. A summary of the material can be found in Section 14.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

INTRODUCTION

SOFTECH

INTRODUCTION

Directives are used to provide supplemental information to the compiler about the program. Directives affect output format, program optimization, data and subroutine linkage, debugging information, and other aspects of program processing.

Most directives change the way a program is processed without changing the computation performed by the program. Perhaps the simplest example of such a directive is "!EJECT," which starts a new page in the compiler's listing of the program.

In general, directives can appear after the reserved word START and before any statement, declaration, or optionally labelled END. Some directives can only be placed in certain positions.

The form is:

```
!directive-name [other-information];
```

All directives begin with an exclamation point and terminate with a semi-colon.

SECTION 2

COMPOOL-DIRECTIVE

COMPOOL-DIRECTIVE

A compool-directive makes declarations available from a compool. The forms are:

$$!COMPOOL\ compool\text{-}file \left\{ \begin{array}{c} name \\ (\ name\ ) \end{array} \right\} \quad , \ldots ;$$

!COMPOOL (compool-file-name) ;

Compool-file is a system-defined name for the file that contains the compool declarations. The form of a compool-file is:

'character ...'

Compool-directives can be given only immediately following START or immediately following another compool-directive.

A compool-file-name enclosed in parentheses implies that all names in the compool are to be made available, except those names used in the compool that were obtained from other compools.

If the compool-directive contains a list of names, only those names will be made available. When the name of a table or block appears in a compool-directive, only the name of the table or block and its attributes given in the heading are made available. If the compool-directive contains a name of a table or block enclosed in parentheses, all of its component names will be made available.

The compool-module must be compiled prior to any module which accesses it. The system may append a suffix on the name of compool-file which may or may not have to be included in the compool-directive.

Consider a DECSYSTEM-10 which if compiled a compool module named TEMP would produce a system file named TEMP.CMP. The compool-directive would have to be of the form:

```
!COMPOOL ('TEMP.CMP'); rather than

!COMPOOL ('TEMP');
```

Example

```
START COMPOOL TEMPLATES;
      TYPE DIMENSIONS
            TABLE;
                  BEGIN
                  ITEM HEIGHT U;
                  ITEM WIDTH U;
                  ITEM LENGTH U;
                  END
      TYPE SEASON STATUS (V(SPRING),V(SUMMER),
      V(FALL),V(WINTER));
TERM

START !COMPOOL('TEMPLATES');
      COMPOOL DATA;
            TYPE COLOR STATUS (V(RED), V(BLUE), V(YELLOW));
            TYPE SIZE TABLE;
                  BEGIN
                  ITEM HEIGHT U;
                  ITEM WEIGHT U;
                  END
            DEF TABLE ROOM(100) DIMENSIONS;
            DEF TABLE CODE (V(YELLOW));
                  ITEM CODENAME C 10;
            DEF ITEM LIGHT COLOR;
TERM
```

Directives	Declarations made available for
!COMPOOL 'DATA'(CODE);	CODE, CODENAME, COLOR
!COMPOOL 'DATA' CODE;	CODE, COLOR
!COMPOOL 'DATA' CODENAME;	CODENAME, CODE
!COMPOOL 'DATA' (ROOM);	ROOM(100)
!COMPOOL 'DATA' LIGHT;	LIGHT, COLOR
!COMPOOL ('DATA');	SIZE, HEIGHT, WEIGHT, ROOM, COLOR, CODE, CODENAME, LIGHT

SECTION 3

COPY-DIRECTIVE

COPY-DIRECTIVE

The copy-directive tells the compiler to copy the text of the named file into an accessing module at the point at which the directive is given.

The form is:

```
!COPY  file-name ;
```

The file name is a character literal, defined by the system.

The copy-directive may be given anywhere in a module.

SECTION 4

CONDITIONAL COMPILATION DIRECTIVES

CONDITIONAL COMPILATION DIRECTIVES

Three directives are defined to provide the capability for conditional compilation.

These forms are:

```
!SKIP [letter] ;

!BEGIN [letter] ;

!END ;
```

The begin-directive begins a text block and identifies it with a given letter. The end-directive delimits the conditional block. The skip-directive identifies the letter associated with the blocks to be skipped.

In the following example, note that the directives themselves are not a part of the compiled program.

Example

Source Program

```
START PROGRAM MAIN;
   BEGIN
   !SKIP Y;
   ITEM RESULT U;
   ITEM COUNT U;
   !BEGIN X;
   REF PROC RND U;;
   RESULT = RND;
   !END;
   !BEGIN Y;
   REF PROC RANDOM U;;
   RESULT = RANDOM;
   !END;
   COUNT = 0;
   CASE RESULT;
      BEGIN
      (DEFAULT);;
      (1:100): COUNT=COUNT+1;
      (101:500):COUNT=COUNT+2;
      (501:900):COUNT=COUNT+3;
      END
   END
TERM
```

Compiled Program

```
START PROGRAM MAIN;
   BEGIN

   ITEM RESULT U;
   ITEM COUNT U;

   REF PROC RND U;;
   RESULT = RND;




   COUNT = 0;
   CASE RESULT;
      BEGIN
      (DEFAULT);;
      (1:100): COUNT=COUNT+1;
      (101:500):COUNT=COUNT+2;
      (501:900):COUNT=COUNT+3;
      END
   END
TERM
```

A skip-directive with a letter skips only the block of text starting with a begin-directive with the same letter.

A skip-directive with no letter skips all blocks of text starting with a begin-directive.

The text following a begin-directive with no letter can be suppressed only a skip-directive with no letter.

SECTION 5

LISTING DIRECTIVES

LISTING DIRECTIVES

The listing directives modify the output listing and may be given anywhere in the module.

The listing-directives are:

!NOLIST;

!LIST;

!EJECT;

!NOLIST suppresses the source listing until the end of the module or until it is re-enabled.

!LIST enables the source listing until the end of the module or until it is suppressed.

!EJECT inserts a page eject in the source listing.

SECTION 6

INITIALIZE-DIRECTIVE

INITIALIZE-DIRECTIVE

The initialize-directive is used to set to zero bits all static data that is not otherwise initialized.

The form is:

```
!INITIALIZE ;
```

The initialize-directive can be given only before a non-nested data declaration. It has effect from the point at which it is given until the end of that scope.

For example:

```
START PROGRAM SAMPLE;
  BEGIN
  ITEM TRAILS F;
  ITEM NAME C 40;
!INITIALIZE;
  ITEM COUNT U;
  TABLE SEC(100);
    BEGIN
    ITEM LENGTH U;
    ITEM WIDTH U = 101(5);
    ITEM HEIGHT U;
    END
  .
  .
  .
  END
TERM
```

TRAILS and NAME have unknown initial values; COUNT and all 101 LENGTHS and HEIGHTS are initialized to all zero bits. All 101 WIDTHS are explicitly preset to 5. Character items following an initialize-directive are also initialized to all zero bits, NOT blank characters.

SECTION 7

ORDER-DIRECTIVE

SOFTECH

ORDER-DIRECTIVE

The allocation-order-directive instructs the compiler to allocate data in the order in which it is given.

The form is:

```
!ORDER ;
```

It can be given only as the first entity in block-body or table entry-description. It only has effect on the table or block in which it is given.

Consider the following table declaration:

```
TABLE PARTS(1000) D;
   BEGIN
   ITEM ID U 5;
   ITEM NUMBER U;
   ITEM FLAG B;
   END
```

The compiler may rearrange the allocation order and conserve storage. However, consider the following table-declaration:

```
TABLE PARTS(1000) D;
   BEGIN
   !ORDER;
   ITEM ID U 5;
   ITEM NUMBER U;
   ITEM FLAG B;
   END
```

The compiler may not rearrange allocation order.

An order-directive may be given in a type-declaration. When the type-name containing the order-directive is used, the order-directive applies to all objects declared of that type.

Given the following table:

```
TABLE PARTS(1000) D;
  BEGIN
  ITEM ID U 5;
  ITEM NUMBER U;
  ITEM FLAG B;
  END
```

The letter D in the table-attributes indicates dense packing. If the compiler is allowed to change the order of allocation, it can allocate ID and FLAG in a single word and conserve storage. (Not all compilers perform this sort of rearrangement.) However, if the programmer wants to be certain that no rearrangement occurs, he can include an allocation-order-directive as follows:

```
TABLE PARTS(1000) D;
  BEGIN
  !ORDER;
  ITEM ID U 5;
  ITEM NUMBER U;
  ITEM FLAG B;
  END
```

SECTION 8

EXPRESSION EVALUATION DIRECTIVES

SOFTECH

EXPRESSION EVALUATION DIRECTIVES

The expression evaluation directives indicate whether or not the compiler is free to rearrange computations within a formula.

The forms are:

!LEFTRIGHT;	Indicates evaluation of operators at the same precedence level must be done from left to right.
!REARRANGE;	Indicates that operators at the same precedence level may be evaluated in any order, as long as the commutative, associative, and distributive laws are observed.

These directives can be given anywhere a directive can be given. The compiler defaults to !REARRANGE.

The effect of an evaluation order directive extends from the point at which it is given to the end of the scope or to the next evaluation directive, whichever comes first.

Given the following formula:

```
HEIGHT*LENGTH*WIDTH
```

If no evaluation-order-directive is given, the compiler can rearrange the formula as follows:

```
LENGTH*HEIGHT*WIDTH
```

Or it can rearrange in any other way to produce efficient code. However, if the leftright-directive is in effect, the compiler must first multiply HEIGHT times LENGTH and then multiply the result by WIDTH.

Consider the following expression:

```
COUNT + SUM + FACTOR
```

This expression is algebraically equivalent to the following expressions:

SUM + COUNT + FACTOR

SUM + FACTOR + COUNT

FACTOR + COUNT + SUM

FACTOR + SUM + COUNT

COUNT + FACTOR + SUM

If a rearrange-directive is in effect, the compiler can use any of the of the above expressions in place of the original expression.

If a leftright-directive is in effect, the compiler can use only the specified expression for evaluation.

> *NOTE: !LEFTRIGHT is most useful when a rearrangement of the operands could cause an overflow or an underflow condition.*

Example

If MAXINTSIZE = 15, MAXINT (15) = $2^{15} - 1 = 32768 - 1$, and the largest integer the compiler will accept is 32767. Consider the following expression:

32767 + (-32767) + 100

The value of the first operand approaches the limit of the compiler. To prevent the compiler from rearranging the evaluation order a leftright-directive may be given before the expression. A rearrange-directive given after the statement returns the evaluation order to the default mode.

SECTION 9

INTERFERENCE-DIRECTIVE

INTERFERENCE-DIRECTIVE

The interference-directive is used to inform the compiler that it cannot assume that the storage for the given names is distinct. The form of the interference-directive is:

!INTERFERENCE data-name : data-name , ... ;

The interference-directive indicates that the storage for the first data-name is not necessarily distinct from the storage for the list of data names following the colon.

The names given in the interference-directive must have been previously declared.

If an interference-directive is not given, the compiler assumes that distinct data names refer to distinct storage and makes optimizations based on that assumption.

The compiler is aware of storage that overlaps because of language features that allow overlaying. However, there are cases in which the compiler is not aware of overlaps and for these cases an interference directive must be given. For example, if two data objects are assigned the same absolute address in different overlay-declarations, an interference-directive should be used to warn the compiler.

An interference-directive can be given only before a declaration.

As an example of the use of the interference-directive, consider the following:

```
TABLE PARTS(10);
      ITEM PARTNO U;
ITEM SIZE F;
ITEM ID F;
OVERLAY POS(3310) PARTS;
OVERLAY POS(3314) SIZE;
!INTERFERENCE PARTS : SIZE, ID;
```

This directive informs the compiler that it should not assume that the storage for PARTS is distinct from the storage for SIZE and ID.

SECTION 10

REDUCIBLE-DIRECTIVE

REDUCIBLE-DIRECTIVE

The reducible-directive is used to allow additional optimizations of function calls. The form is:

```
!REDUCIBLE ;
```

A reducible function is one that has the following characteristics:

- All calls with identically valued actual parameters result in identical function values and output parameter values.
- The only data that is modified by the function call is that data declared within the function.

The compiler can, in some cases, detect the existence of common calls on a reducible function, save the values produced by the first call, delete subsequent calls and use the values produced by the first call.

A reducible-directive is given following the semicolon of the function heading. A reducible function must have the reducible-directive in its definition and all its declarations.

Trigonometric functions are good examples of reducible functions. SIN(ANGLE) always produces the same result for the same value of ANGLE and the function has no side effects.

SECTION 11

REGISTER DIRECTIVES

SOFTECH

REGISTER DIRECTIVES

Register directives are used to affect target-machine register allocation. Three register-directives are defined, namely:

!BASE data-name register-number;

!ISBASE data-name register-number;

!DROP register-number;

Register-number is an integer literal that specifies the register in a target-machine-dependent way.

The base-directive instructs the compiler to load the specified register with the address of the given data-name. This base-directive instructs the compiler to assume that the specified register contains the address of the data object. The drop-directive frees the specified register for other use by the compiler.

Register allocation is not meaningful for all machines. Register directives are ignored for machines that do not use registers.

The register directives may be given anywhere a directive may be given.

> *NOTE: Most often, the optimizer on the compiler can determine the optimal register allocation; register directives should be used carefully and ONLY to fine tune a program.*

SECTION 12

LINKAGE-DIRECTIVE

SOFTECH

LINKAGE-DIRECTIVE

The linkage-directive is used to identify a subroutine that does not have standard J73 linkage.

The form is:

!LINKAGE symbol ...

A linkage-directive can be given only in a subroutine declaration or definition. It is given between the heading and the formal parameter declarations.

For example, a JOVIAL (J73) program may call a subroutine (WRTCEL) written in another language, in this case FORTRAN.

```
START PROGRAM CELSIUS;
      BEGIN
      REF PROC WRTCEL (ICEL, IFAH);
            !LINKAGE FORTRAN;
            BEGIN
            ITEM ICEL S 31;
            ITEM IFAH S 31;
            END
      .
      .
      .
      WRTCEL (CELSIUS, FAHRENHEIT);
      END
TERM
```

The linkage-directive is given in the subroutine-declaration, the REF PROC, to indicate what kind of linkage mechanism is to be used.

SECTION 13

TRACE-DIRECTIVE

SOFTECH

TRACE-DIRECTIVE

The trace-directives are used to follow program execution and monitor data assignments. The trace-directive has one of the following forms:

!TRACE (control) name ,... ;

!TRACE name ,... ;

The first form of the trace-directive is a conditional trace. It causes tracing only if control, which is a Boolean formula, is TRUE. The second form is an unconditional trace.

The names given in the trace-directive are the names to be traced. A name can be a statement name, a subroutine name, or a data name.

- For a statement name, the trace notes each time the associated statement is executed.
- For a subroutine name, the trace notes each call on the subroutine. If the subroutine name given is the subroutine that contains the trace-directive, the trace notes both entry to and exit from the subroutine.
- For a data name, the trace notes any modification of the value of the data object. The new value is included in the trace printout. If the data name is a table, the trace notes any modification of a table item, a table entry, or the entire table. If the data name is a block, the trace notes modification of any enclosed object.

Data names given in the control or as names to be traced must be declared previously. Statement or subroutine names can be declared later.

A trace-directive can be given only before a statement. It applies from the point at which it is given to the end of the scope.

SECTION 14

SUMMARY

SUMMARY

Directives provide supplemental information to the compiler about the program. Directives affect output format, program optimization, data and subroutine linkage, debugging information and other aspects of program processing.

The class and forms are:

Class	Directive Form
compool	!COMPOOL (compool-file); !COMPOOL compool-file name ,... ;
text	!COPY file; !SKIP letter; !BEGIN letter; !END;
listing	!LIST; !NOLIST; !EJECT;
initialization	!INITIALIZE;
allocation-order	!ORDER;
evaluation-order	!LEFTRIGHT; !REARRANGE;
interference	!INTERFERENCE data-name : data-*name* ,... ;
reducible	!REDUCIBLE;
register	!BASE data-name register-number; !ISBASE data-name register-number; !DROP register-number;
linkage	!LINKAGE symbol;
trace	!TRACE (control) name ,... ; !TRACE name ,... ;

THE JOVIAL (J73) WORKBOOK

VOLUME 11

DEFINE CAPABILITY

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 11 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

The JOVIAL (J73) language has a define macro capability which provides the programmer with the ability to do textual substitution at any point in the program. This workbook discusses how these DEFINE's are declared and used. The final section is a summary of the information presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

THE DEFINE-DECLARATION

SOFTECH

DEFINE-DECLARATION

A define-declaration is used to declare the name of a macro and to associate text with that name.

In its simplest form, a define-declaration associates a name with a string. The form is:

DEFINE define-name " define-string " ;

The define-name is a programmer supplied name. The define-string can be any string of characters, including blanks, within enclosing quotes.

The exclamation point (!) and quote character (") both have special meaning with a define-string. Thus, these characters must be doubled to be interpreted literally.

Examples of define-strings:

"3.5 * VELOCITY + EPSILON"	yields	3.5 * VELOCITY + EPSILON
"The CHARACTER ""A"""	yields	THE CHARACTER "A"
"HELP!!"	yields	HELP!

Examples of define-declarations:

```
DEFINE APPROXIMATION "3.5 * VELOCITY + EPSILON";
DEFINE DIRECTIVE "!!COPY 'STATUSDECLS'";
DEFINE LOOP
    "FOR I : 1 BY 1 WHILE I <= 5;
        BEGIN ""FOR LOOP""
        SUM = SUM + VALUE(I);
        IF SUM = 0;
            EXIT;
        END ""FOR LOOP""";
```

The first declaration associates the name APPROXIMATION with a formula. The second define-declaration declares the name DIRECTIVE to be associated with a !COPY-directive. The third define-declaration declares LOOP to expand into a for-loop with comments.

SECTION 2

DEFINE-CALLS

SOFTECH

DEFINE-CALLS

A define-call directs the compiler to make a copy of the define-string associated with the define-name. A define that is declared as:

```
DEFINE  define-name  "define-string";
```

is invoked by simply giving the define-name.

To invoke the define-declaration:

```
DEFINE APPROXIMATION "3 * VELOCITY + EPSILON";
```

simply state:

```
APPROXIMATION
```

When the compiler sees APPROXIMATION it substitutes the associated define-string 3 * VELOCITY + EPSILON. For example:

```
SUM = APPROXIMATION;
```

The compiler would obtain the following after substitution:

```
SUM = 3 * VELOCITY + EPSILON;
```

The compiler only interprets a define-call that is a symbol within the program. It does not process the characters within comments and character literals; a define-call in either of those places is not expanded. A define-call must not appear as a formal parameter to a subroutine or as a part of a subroutine-heading.

SECTION 3

DEFINE-DECLARATIONS AND DEFINE-CALLS WITH PARAMETERS

DEFINE-DECLARATIONS AND DEFINE-CALLS WITH PARAMETERS

A define-declaration may contain parameters, as follows:

```
DEFINE define-name (define-formal, ...) "define-string";
```

The character sequence ",..." indicates that one or more define-formals can be given separated by commas.

A define-formal is a single letter. Within the parenthesized parameter list, define-formals are indicated by that single letter. Within the define-string, define-formals are indicated by that letter preceded by an exclamation point. A define-formal receives its value from the corresponding define-actual given in a call on the define-name.

For example, to provide a convenient notation for incrementation, you can define a name TALLY and associate it with the following string:

```
DEFINE TALLY(A) "!A = !A + 1" ;
```

The define-name TALLY has one define-formal, A, associated with it.

DEFINE-DECLARATIONS -- EXERCISES

Indicate whether the following define-declarations are correct or incorrect.

DEFINE-DECLARATIONS	CORRECT	INCORRECT
DEFINE DEFA(A,B,C,) "!A + !B + 5";		
DEFINE DEFB(ALPHA,BETA) "!ALPHA + !BETA";		
DEFINE A "3 + VELOCITY";		
DEFINE DEFC(A,8,C) "!A + !8 + !C";		
DEFINE CIRC "EXAMPLE: CIRC" ;		
DEFINE TESTA(A,B,C,A,F) "(!A + !B)/ !C = !F";		
DEFINE DTEST(A,B) "!A + !B + !C";		
DEFINE DEFE %THIS IS A COMMENT% "3 * VELOCITY" ;		
DEFINE DEFF "THIS IS A COMMENT" "3 * VELOCITY" ;		

ANSWERS

Indicate whether the following define-declarations are correct or incorrect.

DEFINE-DECLARATIONS	CORRECT	INCORRECT
DEFINE DEFA(A,B,C,) "!A + !B + 5";	X	
DEFINE DEFB(ALPHA,BETA) "!ALPHA + !BETA";		X (A define-formal must be a single letter)
DEFINE A "3 + VELOCITY";	X	
DEFINE DEFC(A,8,C) "!A + !8 + !C";		X (a number is not allowed as define-formal)
DEFINE CIRC "EXAMPLE: CIRC" ;		X (Circular define-declarations are not allowed)
DEFINE TESTA(A,B,C,A,F) "(!A + !B)/ !C = !F";		X (Define-formal must be unique)
DEFINE DTEST(A,B) "!A + !B + !C";		X (Define-string cannot reference undeclared parameter)
DEFINE DEFE %THIS IS A COMMENT% "3 * VELOCITY" ;	X	
DEFINE DEFF "THIS IS A COMMENT" "3 * VELOCITY" ;	X	(NOTE: "THIS IS A COMMENT" will be used as the define-string, and "3*VELOCITY" will be treated as a comment.)

DEFINE-CALLS WITH DEFINE-ACTUALS

Consider the following example:

```
DEFINE SWITCH ( A, B, C, D, E, F )
       "IF !A;
              GOTO !D;
       ELSE
              IF !B;
                     GOTO !E;
              ELSE
                     IF !C;
                            GOTO !F;
                     ELSE
                            GOTO ERROR(5);";
```

The define-name SWITCH is declared with six parameters, the single letters A, B, C, D, E and F. When they are referenced in the define-string, the single letter define-formals are preceded by exclamation points.

A DEFINE that is declared with a define-formal parameter list must be called with a define-actual parameter list. The form is:

define-name [(define-actual , ..)]

The above example could be invoked by:

```
SWITCH( READY, SET, GO, ENABLE, STEADY, TAKEOFF);
```

Result

```
IF READY;
        GOTO ENABLE;
ELSE
        IF SET;
                GOTO STEADY;
        ELSE
                IF GO;
                        GOTO TAKEOFF;
                ELSE
                        GOTO ERROR(5);
```

A define-actual can be any sequence of characters, or a null string. A null string is substituted for every define-actual not named. A define-actual is delimited as follows:

> Use the characters up to and including either the *first right* parenthesis not balanced by a left parenthesis or the first comma that is not within a pair of balanced parenthesis.

Double quote characters can be used to represent an actual not representable by the above rules.

A double quote character is represented by two double quotes in a define-actual.

Examples

```
MAX((,), (), ))
MAX(A, B)
MAX("A, B")
```

Given that the define-name COMPUTE requires three parameters consider the following:

Define-Call	Parameter 1	Parameter 2	Parameter 3
COMPUTE(A,B,C,)	A	B	C
COMPUTE(A(I,J),B,C)	A(I,J)	B	C
COMPUTE(A,,C)	A	(null)	C
COMPUTE(A,B)	A	B	(null)
COMPUTE((A,B),C,D)	(A,B)	C	D
COMPUTE("A,B", C,D)	A,B	C	D

All names in the program that are generated by DEFINE's must be declared.

Generated names, however, cannot be declared using a define-declaration. They must be declared in the usual way.

Example

```
DEFINE NEWNAME(A, B) "ITEM !A$$$!B U 5";

NEWNAME(S, T); yields ITEM S$$$T U 5;
```

Use

```
NEWNAME(S, T) = FACTOR / 12;

yields ITEM S$$$T U 5 = FACTOR / 12;
```

ACTUAL PARAMETERS -- EXERCISES

Indicate the number of actual parameters in each define-call and show what the parameters are:

Define-Call	No.	Parameter 1	Parameter 2	Parameter 3
DISPLAY((A,B,C,))				
DISPLAY(ALPHA,,B)				
DISPLAY((,(,)),(),)				
DISPLAY("A,B",C)				
DISPLAY("""")				
DISPLAY()				

ANSWERS

Indicate the number of actual parameters in each define-call and show what the parameters are:

Define-call	No.	Parameter 1	Parameter 2	Parameter 3
DISPLAY((A,B,C))	1	(A, B, C)		
DISPLAY(ALPHA,,B)	3	ALPHA	null	B
DISPLAY((,(,)),(),)	3	(,(,))	()	null
DISPLAY("A,B",C)	2	A, B	C	
DISPLAY("""")	1	"		
DISPLAY()	0			

SECTION 4

NESTED DEFINE-CALLS

SOFTECH

NESTED DEFINE-CALLS

A define-call may be nested in a define-actual parameter. A define-call that is part of a define-actual is expanded if after the substitution of the define-actual, the define-call is a symbol and not part of a symbol.

Example

```
DEFINE DEF1(A)    "ARG!A = !A";
DEFINE TRIG  "COS";
DEF1(TRIG)  yields ARGTRIG = TRIG
                then   ARGTRIG = COS
```

When the call to DEF1 is encountered with TRIG as the argument, TRIG is substituted for the define-formal, and the resulting text is scanned for any further define calls. The TRIG define-call is encountered and COS is substituted.

Define-calls may also be nested in define-strings. The compiler, in expanding a define-call, first makes a copy of the associated define-string. It then substitutes the actual parameters for the formal parameters. It then examines the resulting string to see if any further expansion can be performed.

Example

```
DEFINE T1(A,B)    "!A/!B**EXP";
DEFINE EXP  "2";
T1(XPOINT,YPOINT) yields XPOINT/YPOINT**EXP
                      then   XPOINT/YPOINT**2
```

When the call to T1 is encountered, the 2 define-actuals are substituted for the define-formals, and the resulting text is scanned for any further define-calls. The EXP define-call is encountered and the associated text is substituted.

DEFINE-CALL -- EXERCISES

Given the following define declarations, expand the define-calls:

```
DEFINE DEFA(A,B,C) "'A:!A, B:!B, C:!C'";

DEFINE DEFB(A) "SINTMDA=!A;";

DEFINE DEFC(B) "(3 + (!B))";

DEFINE DEFD "4";

DEFINE DEFE(A) "'!A'";

DEFINE DEFF(A,B) "'ABC !A DEF !B'";
```

Define-Call	Yields
DEFA("ATEXT")	
DEFA("ATEXT",,)	
DEFA("ATEXT","","CTEXT")	
DEFB(DEFC(DEFD)	
DEFE("""TEST!""")	
DEF((XYZ(,)),MN OP)	

ANSWERS

Define-Call	Yields
DEFA("ATEXT")	'A:ATEXT, B: , C: '
DEFA("ATEXT",,)	'A:ATEXT, B: , C:'
DEFA("ATEXT","","CTEXT")	'A:ATEXT, B: , C:CTEXT'
DEFB(DEFC(DEFD)	SIN+MDA = DEFE(DEFD);
	then SIN+MDA = 3 + (DEFD);
	then SIN+MDA = 3 + 4;
	then SIN+MDA = 7;
DEFE("""TEST!""")	'"TEST!"'
DEFF((XYZ(,)),MN OP)	'ABC (XYZ(,)) DEF MN OP'

SECTION 5

LIST-OPTION

SOFTECH

LIST-OPTION

A define-declaration can also include a list-option, which describes how much information is to be given in the output listing. The general form of the define-declarations is:

DEFINE define-name [(define-formal ,...)] [list-option]

"define-string" ;

The square brackets indicate that both the parenthesized list of define-formals and the list option are optional.

The list option lets you specify whether you want to see the define-string in your program, or the define-call, or both. The list options are:

LISTEXP	Include the expanded define-string in the listing in place of the define-call.
LISTINV	Use the define-call in the listing and do not include the expansion.
LISTBOTH	Include both the define-call and the resulting expansion in the listing.

The exact format of the output listing is implementation dependent.

SECTION 6

SUMMARY

SOFTECH

SUMMARY

The complete form of a define-declaration is:

DEFINE define-name [(define-formal, ...)]

[list-option] " define-string " ;

A define-formal is a single letter. The define-formal is preceded by an exclamation point when used in a define-string. Define-formals are optional.

The list-options are:

LISTEXP	-- List expanded define-string in place of define-call.
LISTINV	-- List the define-call and not the expansion.
LISTBOTH	-- List both the define-call and the expansion.

A define-string has the form:

character ...

The double quote and the exclamation point characters must be doubled to appear in the define-string. A define-string may contain a define-call.

The form of a define-call is:

define-name [(define-actual, ...)]

Define-actual has the form:

character ...

A define-actual is delimited by the first comma not within a pair of balanced parenthesis, an unbalanced right parenthesis, or a pair of double quotes.

A double quote is represented by two double quotes in a define-actual.

A null string is substituted for any missing define-actuals.

A define-actual may be a define-call. A define-call that is part of a define-actual is expanded after the substitution of the define-actual.

SECTION 2

TABLE STRUCTURE AND LAYOUT: SPECIFIED TABLES

SOFTECH

SPECIFIED TABLES

A specified table is one in which each item is explicitly positioned by the programmer.

A specified table may be used in any context in which an ordinary table may be used. It may also be used in a type-declaration to create a template for any number of tables with a particular layout. A specified table is often used to interface with some peripheral device that produces its information in a specified format.

A specified table has the same general form as an ordinary table:

```
TABLE  table-name  table-attributes ;

        entry-description
```

The specified table-kind is given in the table-attributes instead of a packing-spec, as follows:

```
[ dimension-list ] [ structure-spec ] [ table-kind ]
```

The table-kind indicates whether the table has fixed-length entries or variable-length entries. The forms are:

```
W [ entry-size ]

V
```

W indicates that the table has fixed-length entries. Entry-size is an integer formula known at compile-time that gives the number of words each entry occupies for a fixed-length entry table. V indicates that the table has variable-length entries.

POS-CLAUSE

The position of each item in a specified table entry is given by a pos-clause following each item-description in the table, as follows:

```
ITEM  item-name  item-description

        POS ( starbit, startword ) ;
```

Startbit and startword are integer formulae known at compile-time. The first bit of a word is numbered 0; the first word of an entry is numbered 0.

Item positioning must take into account the number of bits in a word. An item that occupies one word or less must not be positioned so that it crosses a word boundary.

> *NOTE: An implementation may restrict legal startbit values for pointers that are initialized.*

Example

```
ITEM  INDEX  U 5 POS(0,1);
```

Index is to be positioned starting at bit zero of word one.

THE * STARTBIT CHARACTER

Every item in a specified table must be positioned. The asterisk character (*) may be used for startbit to indicate that the item should be allocated as if it were declared outside the specified table. In this way, the item may be accessed efficiently.

Given the following declaration for a specified table:

```
TABLE SURVEY (10) W 5;
        BEGIN
        ITEM FLAG B 3 POS (10, 0);
        ITEM HISTORY B 10 POS (0, 0);
        ITEM CASE1 U POS (*, 1);
        ITEM CASE2 U POS (*, 2);
        END
```

The items FLAG and HISTORY are positioned in word 0 of each entry as indicated. The items CASE1 and CASE2 are positioned normally in words 1 and 2 of each entry for efficient usage.

FIXED LENGTH ENTRY TABLES

A specified table with fixed-length entries is indicated by the table-kind W optionally followed by the entry-size. A specified table with fixed-length entries may contain information about the structure and initial values. The form is:

```
TABLE  table-name  [ dimension-list ] [ structure-spec ]
        W  [ entry-size ] [ table-preset ];
                BEGIN
                ITEM  item-name  item-description
                        POS ( startbit , startword ) [ table-preset ];
                ...
                END
```

Examples

1) The following table-declaration may be used to match the format of a particular hardware device:

```
TABLE DEVICE (5) W 2;
        BEGIN
        ITEM CHANNEL1 U 10 POS (0, 0);
        ITEM CHANNEL2 U 10 POS (0, 1);
        END
```

The table is a fixed-length entry specified table having six entries. Each entry occupies two words. The first word of each entry contains the item CHANNEL1 in bits 0 through 9. The second word of each entry contains the item CHANNEL2 in bits 0 through 9. If BITSINWORD is 16, table DEVICE may be diagrammed as follows:

DEVICE

CHANNEL1(0)	word 0
CHANNEL2(0)	word 1
CHANNEL1(1)	word 0
CHANNEL2(1)	word 1

⋮

2) Given the specified table declaration:

```
TYPE COMMUNICATIONLINK (10) PARALLEL W 2;
        BEGIN
        ITEM CODE U POS(0,0);
        ITEM ID U 3 POS(0,1);
        ITEM RANGE S 11 POS(3,1);
        ITEM RATE A 6, 4 POS(3,1);
        END
```

If BITSINWORD is 16, the table can be diagrammed as follows:

CODE(0)	
CODE(1)	
⋮	
CODE(10)	
ID(0)	RANGE(0) RATE(0)
ID(1)	RANGE(1) RATE(1)
⋮	
ID(10)	RANGE(10) RATE(10)

TIGHT STRUCTURE

The entry-size for a specified table with fixed-length entries is given following the W in the table-kind. The entry-size for a specified table with fixed-length entries and tight structure is determined by the bits-per-entry either given or assumed for the structure-spec. If a specified table has tight structure, entry-size must not be given as part of the table-kind.

Given the following table-declarations:

```
TABLE SWITCHES (9) T W;
    BEGIN
    ITEM RDY B POS (0, 0);
    ITEM STAT U 5 POS (1, 0);
    END
```

Each entry contains a one-bit item and a five-bit item. Since the structure-spec does not specify bits-per-entry, the compiler uses the minimum number of bits necessary to represent an entry, six bits.

If BITSINWORD is 16, SWITCHES may be diagrammed, as follows:

SWITCHES

RDY (0)	STAT (0)	RDY (1)	STAT (1)	

.
.
.

RDY (8)	STAT (8)	RDY (9)	STAT (9)	

The starting bit in the pos-clause is assumed to be relative to the start of an entry. The item READY(1) is allocated at bit 6 of the first word. Its position, however, is bit 0 relative to the start of the entry. Bits 12-15 of each word are unused.

Bits-per-entry may be specified so that whole entries of the table are allocated on addressable boundaries. For example, if BITSINWORD is 16 and BITSINBYTE is 8, the following declaration may be written:

```
TABLE SWITCHES (9) T 8 W;
        BEGIN
        ITEM RDY B POS (0, 0);
        ITEM STAT U 5 POS (0, 1);
        END
```

SWITCHES may be diagrammed, as follows:

SWITCHES

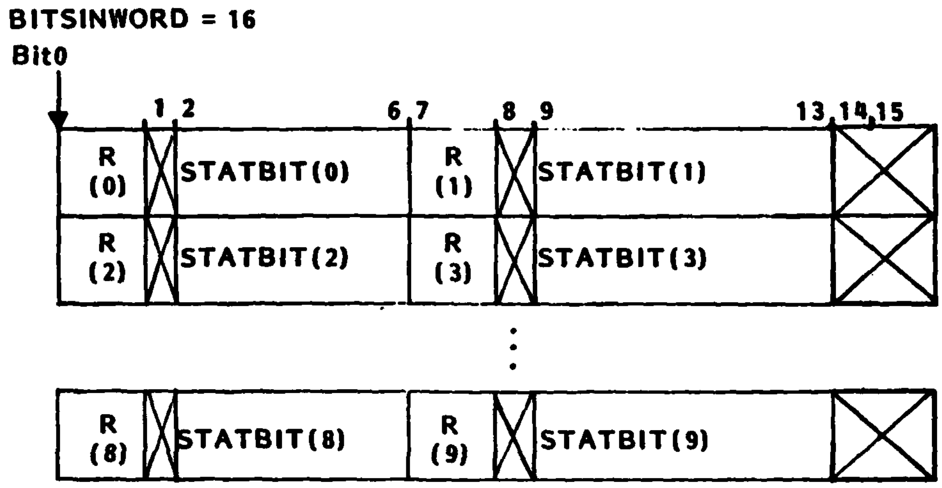
RDY (0)	STAT (0)		RDY (9)	STAT (1)	

⋮

RDY (8)	STAT (8)		RDY (9)	STAT (9)	

As a further example consider the following table diagrammed below:

```
TABLE DRIVER(9) T W;
        BEGIN
        ITEM READY B POS(0, 0);
        ITEM STATBIT U 5 POS(2, 0);
        END
```

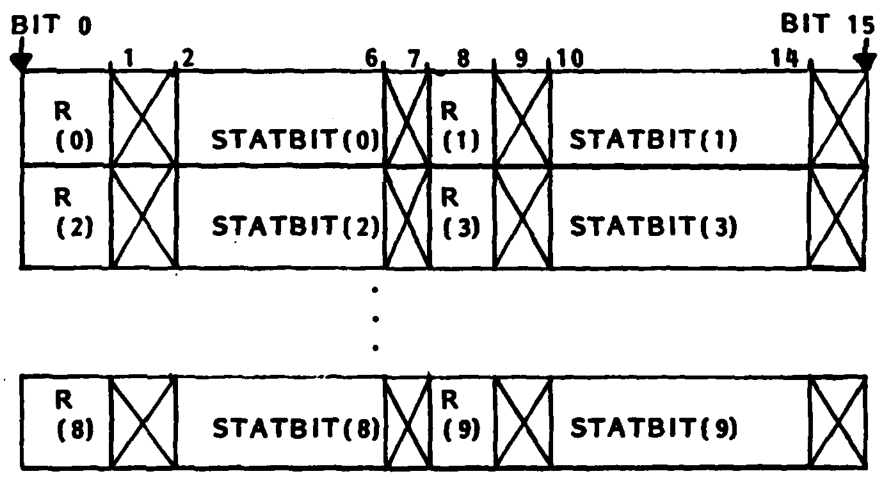


2 entries/word

5 words

If table DRIVER specified 8 bits per entry with tight structure, it would look like this:

```
TABLE DRIVER(9) T 8 W;
      BEGIN
      ITEM READY B POS(0, 0);
      ITEM STATBIT U 5 POS(2, 0);
      END
```



PRESETS

Specified tables may have presets like ordinary tables. If a table-preset is given in the table-attributes, none of the item-declarations within the entry-description may have table-presets. If two items overlap, only one item may be preset.

Examples

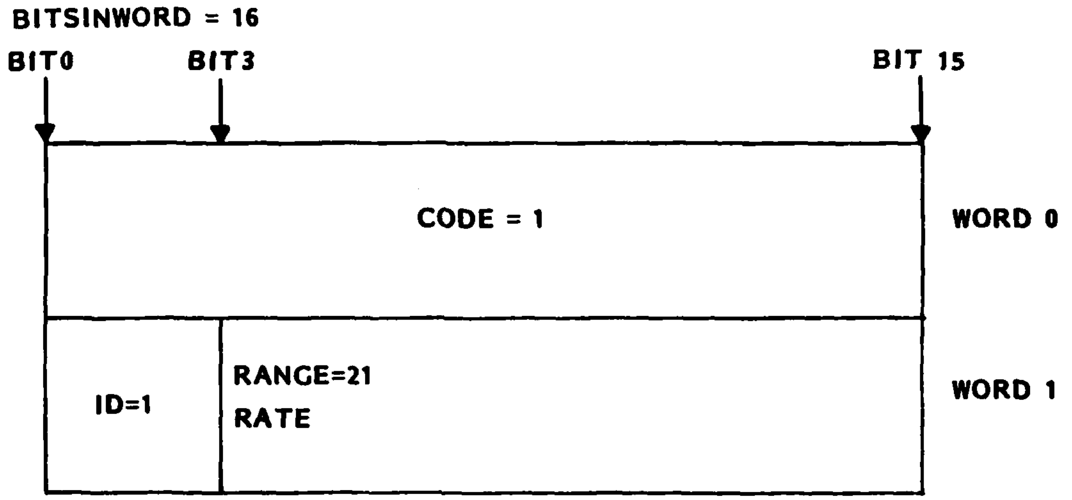
1) Given the following table-declaration:

```
TABLE SPECS (100) W 2 = 2, 4,,,6,8,,,10,12;
          BEGIN
          ITEM LENGTH U POS (0, 0);
          ITEM HEIGHT U POS (0, 1);
          ITEM HIPOINT U 8 POS (0, 1);
          ITEM LOPOINT U 8 POS (8, 1);
          END
```

The items are initialized in order and values are omitted for overlayed items. The value 2 presets LENGTH(0), 4 presets HEIGHT(0). The omitted values prevent HIPOINT and LOPOINT from being initialized. The value 6 presents LENGTH(1), and so on.

2)

```
TABLE COMMUNICATIONLINK (10) W 2 = 11(1,1,21,);
          BEGIN
          ITEM CODE U POS(0, 0);
          ITEM ID U 3 POS(0, 1);
          ITEM RANGE S 11 POS(3,1);
          ITEM RATE A 6,4 POS(3,1);
          END
```



COMMUNICATIONLINK(0)

SPECIFIED TABLES -- EXERCISES

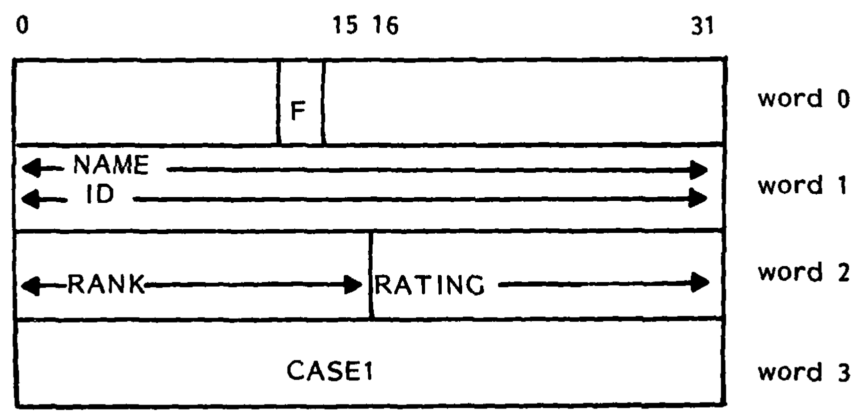
Assuming BITSINWORD is 32 and BITSINBYTE is 8, diagram the following table:

```
TABLE PERSONNEL W 4;
    BEGIN
    ITEM FLAG B 3 POS(10,0);
    ITEM NAME C 4 POS(0,1);
    ITEM RANK C 2 POS(0,2);
    ITEM ID C 4 POS(0,1);
    ITEM RATING C 2 POS(16,2);
    ITEM CASE1 U (*,3);
    END
```

ANSWERS

Assuming BITSINWORD is 32 and BITSINBYTE is 8, diagram the following table:

```
TABLE PERSONNEL W 4;
    BEGIN
    ITEM FLAG B 3 POS(10,0);
    ITEM NAME C 4 POS(0,1);
    ITEM RANK C 2 POS(0,2);
    ITEM ID C 4 POS(0,1);
    ITEM RATING C 2 POS(16,2);
    ITEM CASE1 U (*,3);
    END
```



SPECIFIED TABLES -- EXERCISES

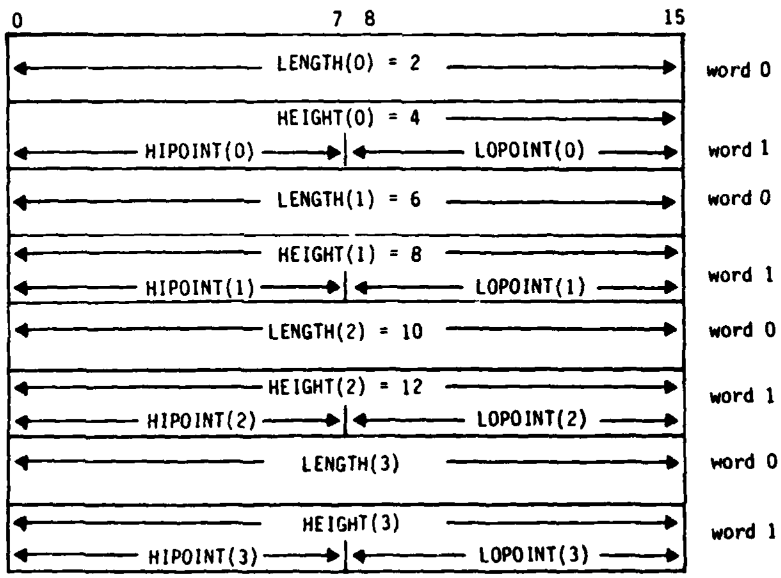
Assuming BITSINWORD is 16 and BITSINBYTE is 8, diagram the following table:

```
TABLE SPECS(10) W 2 = 2,4,,,6,8,,,10,12;
    BEGIN
    ITEM LENGTH U POS(0,0);
    ITEM HEIGHT U POS(0,1);
    ITEM HIPOINT U 8 POS(0,1);
    ITEM LOPOINT U 8 POS(8,1);
    END
```

ANSWERS

Assuming BITSINWORD is 16 and BITSINBYTE is 8, diagram the following table:

```
TABLE SPECS(10) W 2 = 2,4,,,6,8,,,10,12;
    BEGIN
    ITEM LENGTH U POS(0,0);
    ITEM HEIGHT U POS(0,1);
    ITEM HIPOINT U 8 POS(0,1);
    ITEM LOPOINT U 8 POS(8,1);
    END
```



VARIABLE LENGTH ENTRY TABLES

A table with variable-length entries in JOVIAL (J73) is indicated by the table-kind V. Each logical entry of the table may be composed of different numbers of physical entries (words).

A table with variable-length entries provides a way to save space by eliminating unnecessary items from entries, but it is the programmer's responsibility to keep track of where each logical entry begins.

A specified table with variable-length entries may not contain a structure-spec or a table-preset. The form is:

```
TABLE table-name dimension-list V;
      BEGIN
      ITEM item-name item-description
            POS ( startbit , startword ) ;
      ...
      END
```

A physical entry in a table with variable-length entries is one word long. A logical entry in such a table may be composed of many items and may be several words long. The dimensions in a table with variable-length entries determine the number of physical entries in the table. The number of logical entries depends on the way in which the table is built.

A simple but unrealistic example of a table with variable-length entries is the following table:

```
TABLE ALTERNATOR (99) V;
        BEGIN
        ITEM A1 U POS (0, 0);
        ITEM A2 U POS (0, 1);
        ITEM B1 U POS (0, 0);
        ITEM B2 U POS (0, 1);
        ITEM B3 U POS (0, 2);
        END
```

The table ALTERNATOR has two kinds of logical entries; a two word entry (consisting of A1 and A2) and a three word entry (consisting of B1, B2, and B3).

If the table is to have alternating two and three word entries, the first logical entry consists of two words (A1 and A2) and begins at word 0; the second logical entry consists of three words (B1, B2, and B3) and begins at word 2; the third logical entry consists of two words and begins at word 5, and so on.

That is, the table looks as follows:

0	A1 (0)
1	A2 (0)
2	B1 (2)
3	B2 (2)
4	B3 (2)
5	A1 (5)
6	A2 (5)
	...
99	B3 (97)

To locate an item, the beginning of the logical entry is found and the position of the item within that entry is added to that. The next entry is located by adding the number of items in the current entry to the beginning of the current entry

The following may be written to increment A2 in each two-word logical entry and B3 in each three-word logical entry:

```
TWO'WORD = TRUE;

FOR IX ; 0 WHILE IX < 99;

        IF TWO'WORD;

                BEGIN

                TWO'WORD = FALSE;

                A2 (IX) = A2 (IX) + 1;

                IX = IX + 2;

                END

        ELSE

                BEGIN

                TWO'WORD = TRUE;

                B3 (IX) = B3 (IX) + 1;

                IX = IX + 3;

                END
```

This fragment takes advantage of the fact that the logical entries alternate. It uses a flag, TWO'WORD to determine which type of logical entry it is processing. This example is unrealistic because if the entries did alternate as shown, a single five-word entry could be used for each pair of two- and three-word entries. Normally, a logical entry must contain something within it to distinguish it.

A table may be created that contains entries that are two, three, and four words long, as follows:

Two-word-entry	Three-word-entry	Four-word-entry
ENTRY'SIZE	ENTRY'SIZE	ENTRY'SIZE
PART'NUMBER	PART'NUMBER	PART'NUMBER
	ON'HAND	ON'HAND
		DEFECTIVE

ENTRY'SIZE is used to distinguish the different kinds of logical entries. A two-word entry contains ENTRY'SIZE with the value 2 and the number of the part (PART'NUMBER). A three-word entry contains ENTRY'SIZE with value 3, PART'NUMBER, and the number of units currently available (ON'HAND). A four-word entry contains ENTRY'SIZE with the value 4, PART'NUMBER, ON'HAND, and the number of units found to be defective (DEFECTIVE).

An ordinary table with four items in each entry could be used for this table, but two words would then be wasted in entries that only need two words, and one word would be wasted in entries that only need three words.

A table with variable-length entries may be used, as follows:

```
TABLE PARTS (100) V;
        BEGIN
        ITEM ENTRY'SIZE U POS (0, 0);
        ITEM PART'NUM C 5 POS (0, 1);
        ITEM ON'HAND U POS (0, 2);
        ITEM DEFECTIVE U POS (0, 3);
        END
```

Once a program has filled this table with entries, the total number of defective items in the file could be calculated. Each entry in table PARTS that contains a defective item is examined, and the value of DEFECTIVE is added to a counter, COUNT.

Entries that have a DEFECTIVE item are located by the fact that the value of ENTRY'SIZE for an entry with a DEFECTIVE item is 4. The calculation of defective parts is:

```
COUNT = 0;
FOR I : 0 THEN ENTRY'SIZE (I) + I WHILE < 100;
        IF ENTRY'SIZE (I) = 4;
                COUNT = COUNT + DEFECTIVE (I);
```

The for-loop uses ENTRY'SIZE to calculate the position of the next entry in the table. If that entry has four words, it contains a defective unit count, and that count is added to the counter COUNT.

LIKE OPTION WITH SPECIFIED TABLES

A specified table may have fixed or variable length entries. All entries are positioned with a POS clause. When a table-type is declared using a like-option, the following constraints obtain:

1) Entry-size also includes entries declared in the like-option.

2) Table-kind must be the same in the table type and the like-option.

Example

```
TYPE TEST TABLE W 2;
        BEGIN
        ITEM READ B 1 POS (3, 0);
        ITEM SET B 1 POS (0, 1);
        END
TYPE RETEST TABLE
        LIKE TEST W 2;
                ITEM GO B 1 POS (10, 1);
```

SECTION 3

THE OVERLAY DECLARATION

THE OVERLAY DECLARATIONS

The overlay-declaration may be used for allocating several data objects beginning at the same place in storage, for assigning data to a specified machine address, or for specifying the allocation order of a set of data objects.

Examples

OVERLAY LENGTH : DISTANCE;

OVERLAY POS(622) : VELOCITY;

OVERLAY LENGTH, HEIGHT, WIDTH;

A single overlay-declaration may accomplish one or more of these purposes.

The general form of the overlay-declaration is:

OVERLAY [POS (address) :] *overlay-expression;*

An overlay expression is a sequence of one or more overlay-strings separated by colons, as follows:

overlay-string :...

An overlay-string consists of one or more overlay-elements separated by commas, as follows:

overlay-element ,...

An overlay-element is a **name**, a **spacer**, or a **parenthesized overlay-expression**.

> *NOTE: The data objects in an overlay-declaration must all have the same allocation, either static or automatic. An overlay-declaration must not be used to specify more than one physical location for any data object.*

DATA NAMES

The data-names given in an overlay-declaration must be previously declared. They may be the names of items, or entire tables or blocks. They may not be the names of items within a table or the names of items or tables within a block. An overlay-declaration may only name data that is declared in the same scope as the overlay-declaration and which has no REF-specification for it.

Given the following declarations:

```
ITEM COUNT U;

ITEM TIME U;

ITEM MASK B 10;

ITEM RESULT F;

TABLE SPECIFICATIONS (99);

        BEGIN

        ITEM HEIGHT U;

        ITEM LENGTH U;

        ITEM WIDTH U;

        END

TABLE TEST (1 : 50);

        ITEM SUCCESS U;
```

The following overlay-declarations may be written:

```
OVERLAY COUNT : TIME : RESULT;

OVERLAY SPECIFICATIONS : TEST, MASK;
```

The first overlay-declaration contains three overlay-strings. Each string contains one overlay-element. This declaration specifies that the items COUNT, TIME, and RESULT are to be allocated beginning at the same place in storage.

The second overlay-declaration contains two overlay-strings. The first contains one overlay-element, and the second contains two overlay-elements. This declaration specifies that table SPECIFICATIONS is to be allocated beginning at the same place in storage as table TEST. MASK is allocated following TEST, and also shares storage with SPECIFICATIONS. Table SPECIFICATIONS requires 300 words. The first fifty words are shared with table TEST, and the fifty-first word is shared with MASK.

SPACERS

An overlay-element may be a spacer, to indicate how many words to skip over when assigning storage. The form of the spacer is:

W words-to-skip

Words-to-skip is an integer formula known at compile-time that indicates how many words are to be skipped when allocating data in the overlay.

Example

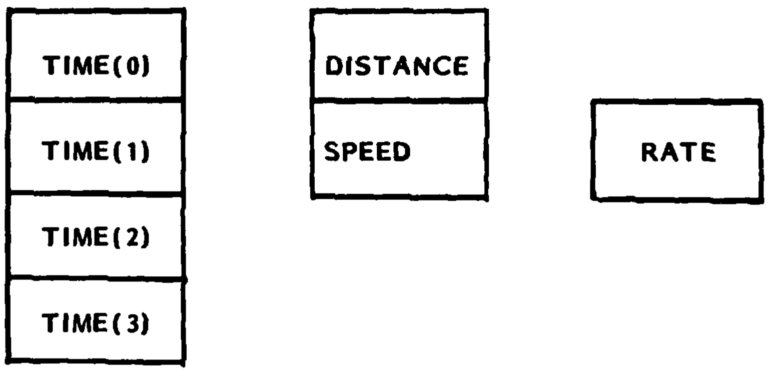
```
TABLE TIMETAB (3);
        ITEM TIME U;
ITEM SPEED U;
ITEM DISTANCE U;

OVERLAY TIMETAB:SPEED, W 2, DISTANCE;
```

TIME(0)	**SPEED**
TIME(1)	
TIME(2)	
TIME(3)	**DISTANCE**

PARENTHESIZED OVERLAY-DECLARATIONS

A parenthesized overlay expression is used to indicate multiple sharing. For instance, in the overlay-declaration below, DISTANCE and SPEED share storage with the first two entries of TIMETAB, and RATE shares storage with SPEED and TIME(1).

```
TABLE TIMETAB(3);
      ITEM TIME U;
ITEM DISTANCE U;
ITEM SPEED U;
ITEM RATE U;
OVERLAY TIMETAB:DISTANCE, (SPEED:RATE);
```



ALLOCATING DATA AT ABSOLUTE ADDRESSES

The overlay-declaration may be used to allocate data at a specific machine address. The form of the overlay-declaration for this case includes a positioner, as follows:

```
OVERLAY  POS  ( address ) : overlay-string :... ;
```

Address is an integer formula known at compile time that gives the decimal address for a word in memory.

The following overlay-declaration may be written to allocate COUNT at machine word 4800:

OVERLAY POS (4800) : COUNT;

A sequence of words may be allocated, as follows:

OVERLAY POS (4800) : COUNT, TIME, SPECIFICATIONS;

Using the declarations given earlier, COUNT is allocated to word 4800, TIME to 4801, and table SPECIFICATIONS to 4802 through 5101.

Storage sharing may be combined with assigning absolute addresses, as follows:

OVERLAY POS (4800) : COUNT : TIME : TEST;

The items COUNT, TIME, and TEST are all allocated at machine address 4800.

An overlay-declaration with an absolute address cannot be given within a block.

ALLOCATION ORDER

An overlay-declaration may be used to specify the order of allocation. Unlike the !ORDER directive, which is used to specify allocation order within a table or block, the overlay-declaration is used to specify order in a more global way.

The following overlay-declaration may be written to allocate the items COUNT, TIME, and TEST in that order:

OVERLAY COUNT, TIME, TEST;

This declaration assures the order of allocation for the three items named in the declaration.

The Overlay Declaration: Examples

1) OVERLAY TIME:SPEED:DISTANCE;

   TIME, SPEED, and DISTANCE all start at the same point in storage.

2) OVERLAY TIME, SPEED:DISTANCE, VELOCITY;

   TIME and SPEED occupy the same storage as DISTANCE and velocity.

3) OVERLAY POS(4880):TIME:SPEED;

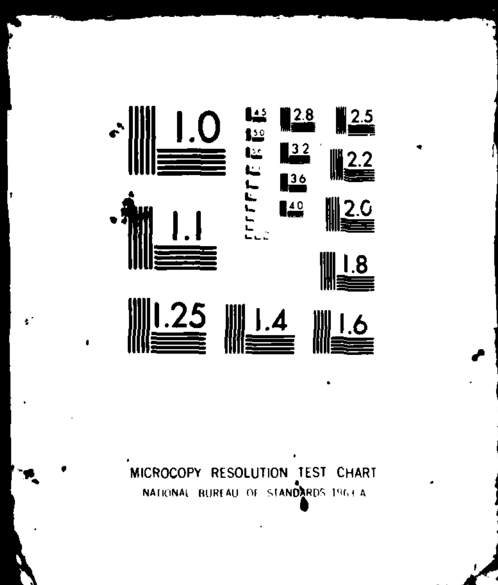
   TIME and SPEED both start at location 4880.

AD-A108 528 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME 10. DIRECTIVES.(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-10 NL

3

2 OF 3

AD-A
108528



OVERLAY DECLARATIONS -- EXERCISES

Given the following declarations:

```
ITEM TIME U;

ITEM SPEED U;

ITEM DISTANCE U;

TABLE DIMENSIONS(2);

        BEGIN

        ITEM LENGTH U;

        ITEM WIDTH U;

        END
```

1. Write a single overlay-declaration that accomplishes the following allocations:

   TIME at 6002

   SPEED at 6004

   DISTANCE at 6002

   DIMENSIONS at 6003

2. Assuming the contents of DIMENSIONS are not used at the same time as the other three items, write an overlay-declaration that is conservative of storage.

ANSWERS

Given the following declarations:

```
ITEM TIME U;

ITEM SPEED U;

ITEM DISTANCE U;

TABLE DIMENSIONS(2);

        BEGIN

        ITEM LENGTH U;

        ITEM WIDTH U;

        END
```

1. Write a single overlay-declaration that accomplishes the following allocations:

TIME at 6002	6002 TIME DIST 6003 DIM
SPEED at 6004	6004 SPEED ↓
DISTANCE at 6002	OVERLAY POS(6002) : TIME, W 1, SPEED : DISTANCE, DIM;
DIMENSIONS at 6003	

2. Assuming the contents of DIMENSIONS are not used at the same time as the other three items, write an overlay-declaration that is conservative of storage.

```
OVERLAY DIMENSIONS : TIME, SPEED, DISTANCE;
```

SECTION 4

SUMMARY

TABLE DECLARATION SYNTAX

The table-declaration syntax, including specified tables, can be written as follows:

```
table-declaration ::=

        TABLE  table-name  [ table-attributes ]

                table-body

table-body ::=

    { ; entry-description                        }
    { table-type-name [ table-preset ];          }
    { unnamed-entry [ table-preset ];            }

unnamed-entry ::=

        type-description  [ { packing-spec } ]
                            { table-kind   }

table-attributes ::=

        [ allocation-spec ] [ ( dimension-list ) ]

        [ structure-spec ] [ { packing-spec } ]
                             { table-kind   }

        [ table-preset ]

structure-spec ::=

        { PARALLEL  |  T  [ bits-per-entry ] }

packing-spec ::=

        { N  |  M  |  D  }

table-kind ::=

        { W [entry-size]  |  V }
```

```
entry-description ::=

     { simple-entry-description   }
     { compound-entry-description }

simple-entry-description ::=

     { table-item-declaration }
     { null-declaration       }

table-item-declaration ::=

     ITEM  item-name  type-description

          [ { packing-spec } ]  [ table-preset ] ;
            { position     }

compound-entry-description

     BEGIN

     simple-entry-description ...

     END
```

THE JOVIAL (J73) WORKBOOK

VOLUME 8

MODULES AND EXTERNALS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 8 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

The workbook discusses the rules for the scope of declarations, the three different kinds of modules in JOVIAL (J73), (main-program-module, compool-module and the procedure-module) and how declarations are communicated from one module to another. The examples presented here are more detailed than those in the videotape. The final section contains a summary of the material presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

<table>
<tr><th>Syntax</th><th>Meaning</th><th>Sample Expansions</th></tr>
<tr><td>[some-feature]</td><td>Brackets indicate an optional feature.</td><td>some-feature
OR
nothing</td></tr>
<tr><td>{one | other}</td><td>Braces with a vertical bar indicate disjunction- a choice between alternatives.</td><td>one
OR
other</td></tr>
<tr><td>{this-one
that-one}</td><td>Braces with each feature on separate lines indicate disjunction - a choice between alternatives.</td><td>this-one
OR
that-one</td></tr>
<tr><td>letter ...</td><td>The sequence '...' indicates one or more repetitions of a feature.</td><td>letter
letter letter
letter letter letter
...</td></tr>
<tr><td>(letter),...</td><td>The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).</td><td>(letter)
(letter) (letter)
(letter) (letter) (letter)
...</td></tr>
<tr><td>[{this-one
that-one}]
+ another</td><td>Syntax symbols may be combined.</td><td>this-one + (another)
that-one + (another)
+ (another)</td></tr>
</table>

SECTION 1

INTRODUCTION

SOFTECH

INTRODUCTION

A complete JOVIAL (J73) program may be written as one module, one compilation unit. It will contain all the declarations, executable statements and subroutines necessary for the execution of the program. See Figure 1-1.

A program may be a collection of more than one module. Each module is created and maintained separately and linked together for execution as a unit. See Figure 1-2.

JOVIAL has three kinds of modules -- main-program modules, procedure modules, and compool modules. The declarations of data objects and subroutines may be placed in a compool-module. This module is used to communicate between the procedure-module and the main-program-module. Subroutine definitions may be placed in a procedure-module. The main program has control over the complete program.

```
START
PROGRAM SEARCH;
        BEGIN
                TYPE KEY STATUS (V(RED), V(GREEN), V(YELLOW));
                TYPE DBASE TABLE (1000);
                        BEGIN
                        ITEM CODE KEY;
                        ITEM VALUE U;
                        END
                ITEM CURVAL U;
                TABLE DATA DBASE;
                GETVALUE (DATA);
                CURVAL = RETRIEVE (V(RED));
                PROC RETRIEVE (ARG1) U;
                        BEGIN
                        ITEM ARG1 KEY;
                        FOR I:0 BY 1 WHILE I <= 1000;
                                IF CODE (I) = ARG1;
                                        RETRIEVE = VALUE (I);
                        ERROR (20);
                        END
        END
        DEF PROC ERROR (ERRNO);
                BEGIN
                ITEM ...
                ...
                END
TERM
```

Figure 1-1. A Complete 1 Module Program

```
START !COMPOOL ( 'DATA' );
      PROGRAM MAIN;
            BEGIN
            FOR I:0 BY 1 WHILE I < UBOUND(PRIVILEGE,0);
                  IF FIND(I,PRIVILEGE) = FIND(I**2,ASSIGNMENT);
                        STOP 21;
            STOP 22;
            END
TERM
      START
      !COMPOOL 'DATA';
      DEF PROC FIND(CODE,TAB);
            BEGIN
            ITEM CODE U;
            TABLE TAB(*);
                  BEGIN
                  ITEM TABCODE U;
                  ITEM TABVALUE F;
                  END
            FIND = -99999.;
            FOR I:0 BY 1 WHILE I<UBOUND(TAB,0);
                  IF CODE = TABCODE(I);
                        BEGIN
                        FIND = TABVALUE(I);
                        EXIT;
                        END
            END
TERM
START COMPOOL DATA;
      DEF TABLE PRIVILEGE (100);
            BEGIN ITEM NUMBER U;
            ITEM NUMBER U;
            ITEM RATING F;
            END
      DEF TABLE ASSIGNMENT(999);
            BEGIN
            ITEM KEY U;
            ITEM COORDINATE F;
            END
      DEF ITEM LIMIT U;
      REF PROC FIND(CODE,TAB) F;
            BEGIN
            ITEM CODE U;
            TABLE TAB(*);
                  BEGIN
                  ITEM TABCODE U;
                  ITEM TABVALUE F;
                  END
            END
TERM
```

Figure 1-2. Multiple Module Program

This main program module (see Figure 1-2) uses the tables declared in the compool module and the function FIND defined in the procedure module and referenced in the compool module. The program consists of the main program module, the compool module DATA and the procedure module.

SECTION 2

DATA ALLOCATION

DATA ALLOCATION

Two kinds of storage are defined for data objects:

Static -- Storage is allocated before program execution and deallocated after program execution. Data objects not declared within subroutines are given static allocation by default. Data objects may be explicitly declared STATIC.

Automatic -- Storage is allocated when the subroutine in which the data object is declared is entered and deallocated upon exit from that subroutine. Data objects declared within subroutines are given automatec allocation by default. Data in subroutines may be allocated statically by use of the STATIC attribute. A data object cannot be explicitly declared AUTOMATIC.

Data may be explicitly declared to be static in item, table, and block-declarations. The forms are:

ITEM name [STATIC] $\left\{\begin{array}{l}\text{type-description}\\ \text{item-type-name}\end{array}\right\}$ [item-preset] ;

TABLE name [STATIC] [(dimension-list)] [table-preset] ;

entry-description

BLOCK name [STATIC] ;

block-body

An item, table, or block declared to be CONSTANT is given static allocation. The STATIC specifier may not be given in a CONSTANT declaration.

A component of a block or table has the same allocation as the block or table of which it is a part. It may not have the STATIC specifier. A component of a block may be a constant only if the block has static allocation.

Only static data may be preset.

Each invocation of a subroutine receives its own copy of automatic variables.

All invocations of a subroutine share variables declared to have STATIC allocation or have STATIC allocation by default.

Example

```
PROC RADICAL (AA, BB, CC) F;
    BEGIN
    ITEM AA F;
    ITEM BB F;
    ITEM CC F;
    ITEM COUNT STATIC U = 0;
    RADICAL = (BB ** 2 - 4. * AA * CC);
    COUNT = COUNT + 1;
    END
```

SECTION 3

DECLARATIONS AND SCOPE

SOFTECH

DECLARATIONS AND SCOPE

DECLARATIONS

The main program module contains declarations. The other kinds of modules, the procedure module and the compool module, also contain declarations. In fact, declarations are an important part of a JOVIAL (J73) program.

A declaration is a "non-executable" construct. That is, it does not represent an action taken when the program is executed. Instead of causing action, each declaration provides information about a name that is used in the program. That information is used by the compiler each time it encounters a use of the declared name.

A declaration does not, in most cases, extend over the entire program. Instead, it applies to a particular part of the program, called the "scope" of the declaration. In fact, the same name can be declared more than once in a program, and each declaration will apply only to its scope. Thus, the programmer does not need to worry about conflicts of names in unrelated parts of a program.

SCOPE

The scope of a declaration is the area in which that declaration applies. A given scope can contain one or more smaller scopes. The number of levels is not limited by the language. In JOVIAL (J73), a scope extends over an entire subroutine. That is, a declaration applies in the entire subroutine in which it is declared, including nested subroutines. Scopes are established during the compilation of a module.

In a subroutine, scope could be diagrammed as follows:

```
PROC MAXVAL (M'IN : M'OUT);
    BEGIN
    ITEM M'IN U;
    ITEM M'OUT U;

    executable statements

    PROC FIGURE (F'IN : F'OUT);
        BEGIN
        ITEM  F'IN F;
        ITEM F'OUT S;

        executable statements

        END
    END
```

Data objects declared in a scope are said to be global to any inner scope. That is, those data objects may be referenced within an inner scope. Using the above example the types and values of the items M'IN and M'OUT are known within the scope of procedure MAXVAL, and thus known within the scope of procedure FIGURE. M'IN and M'OUT are global to procedure FIGURE. At any point in procedure FIGURE, M'IN and M'OUT may be used in a formula or receive an assignment. The types and values of the items F'IN and F'OUT are only known within the scope of procedure FIGURE. MAXVAL may not refer to F'IN or F'OUT. No name is known outside of its scope.

> *NOTE: When a data name is declared in an outer scope and also declared in an inner scope, a reference to that name in an inner scope refers to the inner declaration of that name.*

Example

In Figure 3-1, item SIZE and the procedure names CALCULATE and COMPUTE are in the scope of the procedure module. The names OP1 and OP2 are in the scope of both CALCULATE and COMPUTE. The name RESULT is in the scope of CALCULATE, COMPUTE and SUBTOTAL. A reference to RESULT within SUBTOTAL refers to an output parameter of SUBTOTAL that is an unsigned integer. A reference to RESULT within COMPUTE refers to an output parameter of COMPUTE that is a floating point object.

SCOPE -- LABELS

The scope rules of labels are different than the scope rules for other data objects.

The name of a label is not known in a nested subroutine.

```
PROC COMPUTE;
 +---------------------------------------+
 |  BEGIN                                |
 |  declarations                         |
 |  .                                    |
 |  .                                    |
 |  .                                    |
 |  LAB1: statement;                     |
 |  .                                    |
 |  .  GOTO LAB1; (correct)              |
 |  .                                    |
 |  PROC FIGURE;                         |
 |      +-----------------------+        |
 |      | BEGIN                 |        |
 |      | declarations          |        |
 |      | .                     |        |
 |      | .                     |        |
 |      | .                     |        |
 |      | GOTO LAB1;(incorrect) |        |
 |      | .                     |        |
 |      | .                     |        |
 |      | .                     |        |
 |      | END                   |        |
 |      +-----------------------+        |
 |  END                                  |
 +---------------------------------------+
```

```
START PROGRAM TEST;
 module-scope
     module-body-scope
      BEGIN
         ITEM LENGTH U;
          ...

        PROC CALCULATE (OP1, OP2:RESULT);
          subr-scope
            BEGIN
            ITEM OP1 F;
            ITEM OP2 F;
            ITEM RESULT F;
            ITEM SIZE U;
            ...
            LENGTH = 21;
            ...
            END
        PROC COMPUTE (OP1,OP2:RESULT);
          subr-scope
            BEGIN
            ITEM OP1 F;
            ITEM OP2 F;
            ITEM RESULT F;
            ITEM SIZE U;
            ...

            PROC SUBTOTAL(TOTAL:RESULT);
              subr-scope
                BEGIN
                ITEM TOTAL U;
                ITEM RESULT U;
                ...
                RESULT = TOTAL**2;
                ...

                END

            END
        END
    DEF PROC REPORT(IN,OUT);
        ...
TERM
```

Figure 3-1. Scoping Levels of Main-Program Module

SECTION 4

PROGRAM MODULES

SOFTECH

PROGRAM MODULES

A complete program is made up of one or more modules which are compiled separately and then linked together for execution.

All modules begin with the reserved word START and end with TERM. These words delimit the compilation unit.

There are three kinds of modules: the main-program-module, the compool-module, and the procedure-module. A program must have one and only one main-program-module and may have zero or more compool- or procedure-modules. Each module is created and maintained as a separate text file.

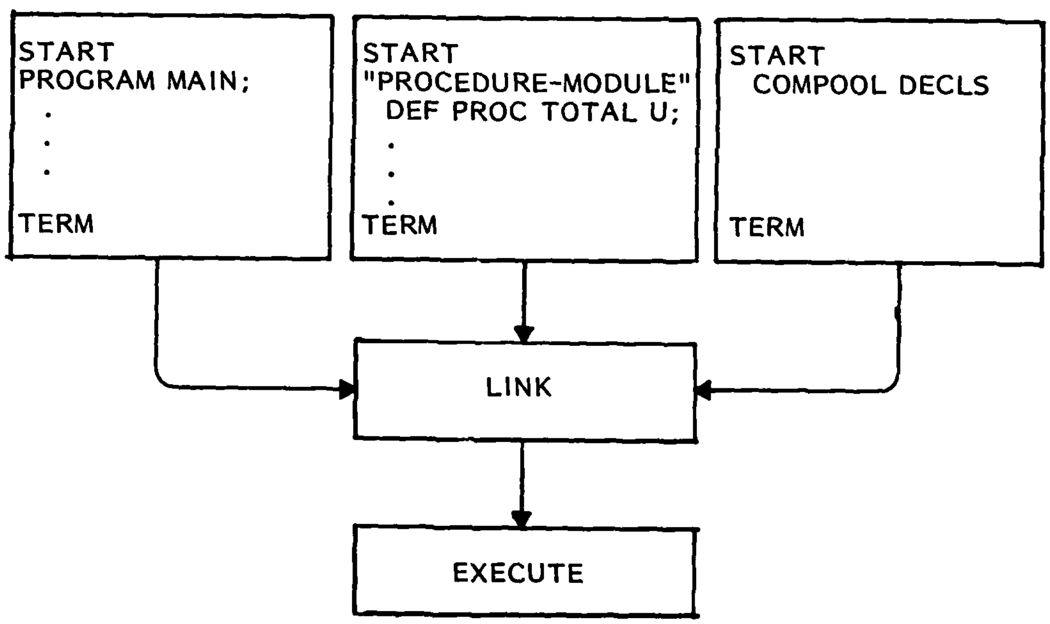


Figure 4-1. A Complete Program

Procedure-modules and compool-modules help in the development of large programs in several ways:

1. When one module is changed and the others are not, only the changed module and the modules it affects need to be recompiled.

2. If the size of the main-program-module exceeds the capacity of a computer, a portion of it can be removed and embodied in a procedure-module. After that, each of the resulting modules is smaller and more likely to fit the machine.

3. When a large project is organized, each main-program-module can be assigned to a specific programmer; program organization can parallel staff organization.

4. Certain modules can be shared among projects. Thus general libraries can be developed.

SECTION 5

THE MAIN PROGRAM MODULE

THE MAIN PROGRAM MODULE

The main-program-module controls the actions to be performed in the complete program. Execution of the program starts at the first statement in the main-program-module and continues until either a stop-statement or the last statement in the main-program-module is reached. Execution may include calls to subroutines that may have been compiled separately in procedure-modules (see Section 6) and imported through a compool-module (see Section 7).

A main-program-module may not have parameters, nor may it be called by another subroutine.

A main-program-module contains a program-body and an optional sequence of non-nested subroutines. The form of the main-program-module is:

```
START PROGRAM name;
    BEGIN
    [ declaration ]
    ...
    executable statements
    ...
    [ subroutine-definition ]
    ...
    END
    [ subroutine-definition ]
    ...
TERM
```

The declarations and subroutine-definitions are optional, but the program-body must contain at least one executable statement.

A non-nested subroutine is a subroutine definition that can be made external by the addition of the DEF reserved word, as follows:

[DEF] subroutine-definition

A non-nested subroutine can contain nested subroutines.

Consider the following main-program-module:

```
START PROGRAM SEARCH;
        BEGIN
        TYPE KEY STATUS (V(RED), V(GREEN), V(YELLOW));
        TYPE DBASE
                TABLE (1000);
                        BEGIN
                        ITEM CODE KEY;
                        ITEM VALUE U;
                        END
        TABLE DATA DBASE;
        ITEM CURVAL U;
        GETVALUE(DATA);
        CURVAL=RETRIEVE(V(RED));
        PROC RETRIEVE(ARG1) U;
                BEGIN
                ITEM ARG1 KEY;
                FOR I:0 BY 1 WHILE I<=1000;
                        IF CODE(I) = ARG1;
                                RETRIEVE = VALUE(I);
                ERROR(20);
                END
        END
```

```
DEF PROC GETVALUE(ARGTAB);
      BEGIN
      TABLE ARGTAB DBASE;
      ...
      END
DEF PROC ERROR(ERRNO);
      BEGIN
      ITEM ERRNO U;
      ...
      END
TERM
```

This main-program-module consists of a program-body and two non-nested subroutines. The program-body contains two type-declarations , a table-declaration, an item-declaration, two statements, and a nested subroutine-definition.

This main-program-module is independent and could be compiled and executed. Data declared in the main-program-module is given STATIC allocation by default.

SECTION 6

PROCEDURE MODULES AND EXTERNALS

SOFTECH

PROCEDURE-MODULES AND EXTERNALS

PROCEDURE MODULES

A procedure-module provides a way in which the subroutines of a program can be compiled separately. A procedure-module contains declarations and subroutine-definitions, as follows:

```
START
        [ declaration ... ]
        [  [ DEF ]  subroutine-definition ... ]
TERM
```

As an example of a procedure-module, consider the following:

```
START !COMPOOL ('TYPEDEFS');
        DEF PROC GETVALUE(ARGTAB);
                BEGIN
                TABLE ARGTAB DBASE;
                  ...
                END
        DEF PROC ERROR(ERRNO);
                BEGIN
                ITEM ERRNO U;
                ...
                END
TERM
```

The procedure module contains two external subroutine definitions. The type-name DBASE is provided by the declaration of DBASE in the compool TYPEDEFS.

Since the main-program-module is compiled at one time and the procedure-module is compiled at another, the subroutine-names must be made known external to the procedure-module and the subroutine-names must be properly communicated to the modules that call them.

EXTERNAL DEFINITIONS

The subroutine-definitions must be brought into the scope of the calling module. This is done by means of external definitions.

There are two kinds of external-declarations:

DEF-specification

> to make a name available outside the scope in which it is declared (exporting a name)

REF-specification

> to bring into a scope a name DEF'ed in another module (importing a name)

> *NOTES: An external-declaration can be used to make a data name declared in one module accessible to other modules.*
>
> *All external data names must be distinct throughout the complete program.*
>
> *External-declarations of formal parameters is not permitted.*

DEF-SPECIFICATION

A DEF-specification specifies that a name is available outside of the scope of which it is declared. The form is:

```
DEF declaration

DEF BEGIN
       declaration ...
       END
```

```
DEF PROC name [ use-attribute ]
        [ ( formal-list ) ]  [ item-type-description ];
                subroutine-body
```

The following is an example of a DEF-specification:

```
DEF BEGIN
        ITEM RATE U 10;
        ITEM TIME U 15;
        TABLE STOCKS (100);
                BEGIN
                ITEM NAME C 6;
                ITEM QUOTE C 3;
                END
        END
```

This external-declaration declares the items RATE and TIME and the table STOCKS. All of those data objects and their attributes are "exported" outside the module containing the DEF-specification.

A DEF-specification can be used to declare an item, table, block, or statement name. A DEF-specification can be used to define a subroutine in a main program or procedure module, but not in a compool module.

A DEF-specification for a statement name makes the address of the statement available for linkage purposes. The statement name, however, cannot be used as the target of a GOTO statement that is in another module, or in any other way to cause control to transfer outside the given scope.

Data or subroutines declared by a DEF-specification in a module are physically allocated in that module.

A DEF-specification can only be used with data objects that are allocated statically. Data declared external in a subroutine, therefore, must have a STATIC allocation-spec. Data objects may be preset.

For example, to declare the external item FLAGS within the procedure MONITOR, one can write:

```
PROC MONITOR(STATE);
        BEGIN
        DEF ITEM FLAGS STATIC B 5;
        ...
        END
```

The item FLAGS is declared as an external name. The declaration includes the STATIC allocation-spec because the declaration is given within the subroutine MONITOR.

If a subroutine-definition in a procedure-module is preceded by DEF, that subroutine may be invoked from within the main-program-module or from within another procedure-module, provided that the referencing module contain an appropriate REF-specification for the subroutine for accesses a compool containing such a specification.

REF-SPECIFICATION

The REF-specification imports a name and its attributes which was declared by a DEF-specification in another module.

The forms are:

```
REF declaration

REF BEGIN
        declaration ...
        END

REF PROC name [ use-attribute ]
        [ ( formal-list ) ]  [ item-type-description ] ;
                parameter-declarations
```

A name declared in a REF-specification must agree in name, type and all other attributes with the name declared in the corresponding DEF-specification.

A REF specification can be used to import information about items, tables, blocks or subroutines.

A constant item or table cannot appear in a REF-specification.

A REF-specification may be given for a block containing a constant declaration. The following example is the only case in which a preset may be given in a REF-specification.

```
DEF BLOCK PSEUDOBLOCK;
        BEGIN
        CONSTANT ITEM P1 F = 3.14159;
        END
```

Example - DEF-REF Communication

```
START
PROGRAM XX;
        BEGIN
        REF ITEM YY U;
        REF PROC ZZ;
                BEGIN       END
        REF PROC WW (V1 : V2);
                BEGIN
                ITEM V1 F;
                ITEM V2 F;
                END
```

```
        ...
        WW(IN : OUT);
        ...
        ZZ;
        ...
        YY = YY + 1;
        ...
        END
TERM
START
DEF ITEM YY U;
DEF PROC ZZ;
        BEGIN
        ...
        PROC AA;
        END
DEF PROC WW (V1 : V2);
        BEGIN
        ITEM V1 F;
        ITEM V2 F;
        ...
        END
TERM
```

SECTION 7

COMPOOL-MODULES

COMPOOL-MODULES

The third kind of module is called a compool, (from the original "common declaration pool"). A compool-module may be used to communicate between separately compiled-modules. A compool contains declarations of global data (DEFs), references to procedures declared in another module (REFs), constant data objects, and type-declarations. The form of a compool-module is:

```
START
        COMPOOL compool-name;
                compool-declaration ...
TERM
```

The following kinds of declarations are allowed in a compool-module:

- constant-declaration
- type-declaration
- define-declaration
- overlay-declaration
- DEF-specification for a data or statement name declaration
- REF-specification for a data or subroutine declaration

As an example of a compool-declaration, consider the following:

```
START COMPOOL TYPEDEFS;
        TYPE KEY STATUS (V(RED), V(GREEN), V(YELLOW));
        TYPE DBASE
                TABLE (1000);
                        BEGIN
                        ITEM CODE KEY;
```

```
            ITEM VALUE U;
            END
        TERM
```

The compool TYPEDEFS contains two type-declarations, one for the item type KEY and one for the table type DBASE.

COMPOOL DIRECTIVES

The information in a compool-module is made available to the module being compiled by a compool-directive. Compool-directives are given immediately following the START in the module being compiled, or following another !COMPOOL-directive.

The forms are:

```
!COMPOOL (compool-file);

!COMPOOL  compool-file  name, ...;

!COMPOOL  compool-file  (name), ...;
```

Compool-file name is a character-literal.

A compool-file enclosed in parentheses implies all names in the compool are to be made available, except those names used in the compool that were obtained from other compools.

If the compool-directive contains a list of names, only those names will be made available.

If the compool-directive contains a name of a table or block enclosed in parentheses, all of its component names will be made available.

The names given in a compool-directive must be declared in the designated compool. A name may not be a component of a type-name or the name of a formal parameter.

If the name given in a compool-directive is the name of an item, table, or block declared using a type-name, then the declaration of the type-name is also made available, provided it is declared within the compool and not brought in by a compool-directive.

For a pointer item, the definition of the type-name that is the pointed-to type is also made available, provided it is declared within the compool and not brought in by a compool-directive.

If the given name is the name of an item within a table, then the table name is also made available.

If the given name is a table name, the definitions of any status-lists or status-type-names associated with the table's dimensions are also made available, provided they are declared in the compool.

If the given name is a table type-name or block type-name, the definitions of the components are made available.

If the given name is a status item name, its associated status-list and status type-name (if any) are also made available, provided they are declared in the compool.

If the given name is the name of a subroutine, any type-names associated with the subroutine's formal parameters or return value are also made available, provided they are declared in the compool.

Examples

Consider the following compool:

```
START COMPOOL BSQDATA;
      DEF ITEM HEIGHT U;
      DEF ITEM WIDTH U;
      DEF ITEM LENGTH U;
      DEF TABLE GRID (20,20);
            BEGIN
            ITEM XCOORD U;
            ITEM YCOORD U;
            END
      TERM
```

The following list gives different forms of the compool-directive and indicates the declarations that are made available for each form.

Directive	Available Declarations
!COMPOOL 'BSQDATA' LENGTH;	LENGTH
!COMPOOL 'BSQDATA' LENGTH, WIDTH;	LENGTH, WIDTH
!COMPOOL 'BSQDATA' GRID	GRID
!COMPOOL 'BSQDATA' (GRID);	GRID, XCOORD, YCOORD
!COMPOOL ('BSQDATA');	LENGTH, HEIGHT, WIDTH, GRID, XCOORD, YCOORD

> *NOTE: The compool 'BSQDATA' must be compiled before the module which would access it through a !COMPOOL directive is compiled.*

Compool-modules and declarations must not be used in a circular way. The following is an illegal usage of compool directives:

```
START !COMPOOL('TYPEDESCR');
      COMPOOL DECLS;
            DEF TABLE ID(1 : 5, 2 : 6);
                  BEGIN
                  ITEM NAME LONGSTR;
                  ITEM BRIBE VALUE;
                  ITEM GRADE LETTER
                  END
            TYPE MAXVAL U 15;
TERM
```

```
START !COMPOOL ('DECLS');
      COMPOOL TYPEDESCR;
            TYPE LETTER STATUS (V(A), V(B), V(C), V(D));
            TYPE LONGSTR C 20;
            TYPE VALUE U 18;
            DEF ITEM NUMBER MAXVAL;
TERM
```

COMPOOL-DIRECTIVE -- EXERCISE

Given the following compool-module:

```
START
    COMPOOL DATA;
        DEF ITEM TEST F;
        DEF TABLE SQUARE (1 : 10);
            BEGIN
            ITEM LENGTH U;
            ITEM WIDTH U;
            END
        DEF ITEM COUNT S;
TERM
```

and the following compool-directives, indicate the information available to the module containing the compool directive:

Directive	Information
!COMPOOL 'DATA' TEST;	
!COMPOOL 'DATA' TEST, COUNT;	
!COMPOOL 'DATA' SQUARE;	
!COMPOOL 'DATA' (SQUARE), COUNT;	
!COMPOOL ('DATA');	

ANSWERS

Directive	Information
!COMPOOL 'DATA' TEST;	TEST
!COMPOOL 'DATA' TEST, COUNT;	TEST, COUNT
!COMPOOL 'DATA' SQUARE;	SQUARE (1 : 10)
!COMPOOL 'DATA' (SQUARE), COUNT;	SQUARE (1 : 10), LENGTH, WIDTH COUNT
!COMPOOL ('DATA');	TEST, SQUARE (1 : 10), LENGTH, WIDTH, COUNT

SECTION 8

MODULE COMMUNICATION

MODULE COMMUNICATION

Modules may communicate by using compool-directives as shown in the preceding sections. If a declaration is to be used in more than one module, it is passed in a compool. It may be referenced in each module that needs it by using a compool-directive.

As an example of communication by using a compool-directive consider:

```
START COMPOOL TYPEDEFS;
        TYPE DBASE TABLE (1000);
                BEGIN
                ITEM CODE KEY STATUS (V(RED), V(GREEN));
                ITEM VALUE U;
                END
TERM
START !COMPOOL ('TYPEDEFS');
        COMPOOL DATABASE;
        DEF TABLE DATA DBASE;
TERM
```

The compool-module TYPEDEFS contains the type-declaration for the table type DBASE. The compool-module DATABASE contains the compool-directive, which makes the declarations of the compool TYPEDEFS available. Thus, the type-name DBASE does not have to be declared in the module.

If communication between modules is accomplished through compool-directives, the compiler provides the declaration of the shared object. If the module using that object does not use it in a manner that is consistent with its declaration, the error is detected and reported at compile-time.

A REF-specification can be used in one module to directly communicate with another module, but in this case, no checking can be performed. The compiler assumes that the type class and attributes given in the REF-specification are accurate. At link time, the references to the name are bound together, but no check of type or attributes can be made because that information is no longer available.

Thus, if the REF-specification declares an object of one type and the DEF-specification declares an object of another type, the program that is formed by linking the separately compiled modules is invalid and the results of its execution are unpredictable.

As an example of direct communication, consider a procedure module which contains some external subroutine definitions as follows:

```
START !COMPOOL (TYPEDEFS);
        DEF PROC GETVALUE(ARGTAB);
                BEGIN
                TABLE ARGTAB DBASE;
                 ...
                END
        DEF PROC ERROR(ERRNO);
                BEGIN
                ITEM ERRNO U;
                ...
                END
TERM
```

Now, suppose that the module DATABASE does not contain a REF-specification for the subroutine ERROR, but instead the main-program module includes a REF-specification for ERROR, as follows:

```
START !COMPOOL (DATABASE);
      PROGRAM SEARCH;
            BEGIN
            REF PROC ERROR(ERRNO);
                  ITEM ERRNO U;
            GETVALUE(DATA);
            CURVAL = RETRIEVE(V(RED));
            PROC RETRIEVE(ARG1) U;
                  BEGIN
                  ITEM ARG1 KEY;
                  FOR I:0 BY 1 WHILE I <=1000;
                         IF CODE(I) = ART1;
                               RETRIEVE = VALUE(I);
                  ERROR(20);
                  END
            END
TERM
```

In this case, the REF-specification for ERROR agrees with the DEF-specification, and the resulting program operates correctly. However, suppose the REF-specification indicated that the subroutine ERROR has two arguments. The compiler cannot detect any error, the linker makes the connection and the resulting program is invalid but no indication of its invalidity can be made.

Once all modules have been compiled, they may be linked together and run as a complete program. If a change is made in the main-program-module, that is the only module which would need to be recompiled since no other module accesses it. If a change is made to a compool-module, it and all modules which access it must be recompiled.

SECTION 9

SUMMARY

SUMMARY

DATA ALLOCATION

Allocation of storage for a data object can be STATIC or automatic. STATIC allocation means that the data object is to exist throughout the entire execution of the program. Automatic allocation is applicable only to data declared within subroutines and means that the data object need only exist while the subroutine is executing (i.e., values are not necessarily preserved between calls). Automatic is the default allocation for data declared in subroutines and cannot be explicitly specified. STATIC is the default for data not declared in subroutines and can be explicitly specified both inside and outside of subroutines.

SCOPE

The scope of a declaration is the text over which the declaration may be legally accessed.

PROGRAM-MODULES

A complete program is made up of one or more modules. Each module is compiled separately and later linked together. A module is delimited by START at the beginning and TERM at the end.

There are three kinds of modules. They are:

- main-program-module
- compool-module
- procedure-module

A program must have one and only one main-program-module and may have zero or more compool- or procedure-modules.

MAIN PROGRAM MODULE

The main-program-module controls the actions to be performed. Only one may exist in a complete program.

A form of a main-program-module is:

```
START
        PROGRAM  program-name;
                program-body
                [ subroutine-definition ... ]
TERM
```

PROCEDURE MODULES

A procedure-module is to define global subroutines. Those subroutines may then be accessed in other modules. A form of a procedure-module is:

```
START
        [ declaration ] ...
        [ subroutine-definition ] ...
TERM
```

EXTERNAL DECLARATION

An external-declaration can be used to make data name declared in one module accessible to other modules. There are two kinds of external-declarations:

DEF - specification

REF - specification

DEF-SPECIFICATION

A DEF-specification specifies that a name is available outside of the scope which is declared. The form is:

```
DEF declaration

DEF BEGIN

    declaration ...

    END

DEF PROC name [ use-attribute ]

    [ ( formal-list ) ]  [ item-type-description ] ;

        subroutine body
```

REF-SPECIFICATION

A REF-specification references an external name. The forms are:

```
REF declaration

REF BEGIN

    declaration ...

    END

REF PROC name [ use-attribute ]

    [ ( formal-list ) ]  [ item-type-description ] ;

        parameter-declarations
```

COMPOOL-MODULES

A compool-module provides for the communication of names between separately compiled modules. A compool-module can contain only declarations. The form is:

```
START  COMPOOL  compool-name ;

    declaration ...

TERM
```

COMPOOL-DIRECTIVES

A compool-directive is used to identify the compool and the set of names from that compool that are to be used in the compool scope for the module being compiled. The forms are:

```
!COMPOOL (compool-file);

!COMPOOL compool-file name, ...;

!COMPOOL compool-file (name), ...;
```

MODULE-COMMUNICATION

Compool-modules are compiled prior to any module which would access it. Procedure-modules and the main-program module are then compiled. The main-program-module calls in the compool to access global data items and the externally known subroutine.

The modules communicate by using a compool-directive or a REF-specification in one module to directly communicate with another module.

THE JOVIAL (J73) WORKBOOK

VOLUME 9

ADVANCED TOPICS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 9 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

The layout of ordinary and specified tables is discussed in Sections 1 and 2. The Overlay declaration is addressed in Section 3. Section 4 is a syntactic summary of the material presented in Sections 1 and 2.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
$\left\{\begin{matrix}\text{this-one}\\\text{that-one}\end{matrix}\right\}$	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
$\left[\left\{\begin{matrix}\text{this-one}\\\text{that-one}\end{matrix}\right\}\right]$ + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

TABLE STRUCTURE AND LAYOUT: ORDINARY TABLES

SOFTECH

TABLE STRUCTURE AND LAYOUT: ORDINARY TABLES

Workbook 9 considers some advanced features of JOVIAL (J73). Successful programs can be written without using any of these features, but they become increasingly necessary when memory space is limited, or when communicating with hardware devices that produce and expect data in certain formats. This section examines the structure and layout of JOVIAL (J73) ordinary tables. (Ordinary tables were discussed in Workbook 4). The other type - specified tables - will be discussed in the next section.

An ordinary table is one for which the compiler determines the storage layout from packing and structure information specified in the table - declaration. The packing-spec describes the way in which items within an entry are packed. The structure-spec describes the structure of the table in memory (serial or parallel) or the number of entries to be allocated per word (tight structure).

PACKING

Table packing refers to the allocation of items within an entry to words of storage. If a table entry contains more than one item, the way in which the items of the entry are packed can be specified by giving a packing-spec.

The packing-spec may be given as part of the table-declaration, as follows:

```
TABLE  table-name  [ ( dimension-list ) ] [ packing-spec ] ;
           entry-description
```

The square brackets indicate that the dimension-list and the packing-spec are optional.

A packing-spec may be given for any item in the table, as follows:

ITEM item-name item-description [packing-spec];

If the packing-spec is given in the table-attributes, it applies to the entire table. All items are packed according to that packing-spec except those items that have a packing-spec in their declaration.

> *NOTE: Some implementations will reorder items within an entry for purposes of more efficient packing unless the !ORDER directive is in effect.*

The packing-spec is one of the following:

N No packing. Each item is allocated in a new word. (This is the default.)

M Medium packing. The amount of packing depends on the implementation.

D Dense packing. The compiler packs as many items of an entry as possible within a word, making use of all available bits within the word. Items that occupy one word or more are always allocated at a word boundary and bytes of a character item are always aligned on a byte boundary.

If a packing-spec is not given, the compiler assumes N (no packing) for all tables except for those with tight structure, which is described later in this section.

Examples

1) Given the following table-declaration:

```
TABLE TRACK (1 : 100);
        BEGIN
        ITEM DIST U 5;
        ITEM SB B 3;
        ITEM ANGLE S 10;
        END
```

The compiler assumes TRACK is a serial table with no packing and allocates each item to a separate word. If BITSINWORD is 16, TRACK may be diagrammed as follows:

TRACK

DIST(1)	
SB(1)	TRACK(1)
ANGLE(1)	

...

DIST(100)	
SB(100)	TRACK(100)
ANGLE(100	

Table TRACK requires 300 words of storage.

2) Given a table-declaration for the same table that includes a packing-spec of D:

```
TABLE TRACK (1 : 100) D;
      BEGIN
      ITEM DIST U 5;
      ITEM SB B 3;
      ITEM ANGLE S 10;
      END
```

The compiler packs as many items of a single entry as possible within a word. The total number of bits required for each entry is 19. If BITSINWORD is 16, the compiler packs DIST and SB into one word, using two words for each entry. TRACK may be diagrammed as follows:

TRACK

DIST(1)	SB(1)	TRACK(1)
ANGLE(1)		

...

DIST(100)	SB(100)	
ANGLE(100)		

Table TRACK, requires 200 words of storage.

If BITSINWORD is 32, the compiler is able to pack all three items of an entry into a single word. That layout may be diagrammed as follows:

TRACK

DIST(1)	SB(1)	ANGLE(1)	TRACK(1)

...

DIST(100)	SB(100)	ANGLE(100)	TRACK(100)

Table TRACK requires 100 words of storage.

3) Given a table-declaration for the same table that includes a packing-spec of D in the table-attributes and a packing-spec of N in the item-declaration of SB:

```
TABLE TRACK (1 : 100) D;
        BEGIN
        ITEM DIST U 5;
        ITEM SB B 3 N;
        ITEM ANGLE S 10;
        END
```

The packing-spec for the table indicates dense packing, but the packing-spec for item SB indicates no packing. All other items in the table may be packed densely, but item SB must occupy a word by itself.

If the given implementation reorders items within an entry and an !ORDER directive is not in effect, DIST and ANGLE may be packed into one word and SB allocated in another word. If BITSINWORD is 16, TRACK may be diagrammed as follows:

TRACK

DIST(1)	ANGLE(1)	TRACK(1)
SB(1)		

...

DIST(100)	ANGLE(100)	TRACK(100)
SB(100)		

If the implementation does not perform reordering or an !ORDER directive is in effect, the items are each allocated a new word and the table requires 300 words of storage.

4) Given a table-declaration without a packing-spec, N (no packing) is assumed. Several items within the table may have a packing-spec of D, as follows:

```
TABLE SUPERTRACK (100);
    BEGIN
    ITEM DIST U 5;
    ITEM SB B 3 D;
    ITEM ANGLE S 10;
    ITEM MASK1 B 4 D;
    ITEM MASK2 B 2 D;
    END
```

This declaration directs the compiler to allocate a separate word for DIST and a separate word for ANGLE and to pack MASK1 and MASK2 within a single word, with the option of either packing SB with MASK1 and MASK2 or allocating a separate word for SB. If the implementation of the compiler performs reordering and an !ORDER directive is not in effect, SB, MASK1, and MASK2 may be packed in the same word. If the compiler does not perform reordering or an !ORDER directive is in effect, SB will be allocated a separate word.

PACKING -- EXERCISES

Given the following two table declarations:

```
TABLE TRACK1 (1:100);        TABLE TRACK2 (1:100);
     BEGIN                        BEGIN
     ITEM DISTANCE U 5;           ITEM DIS2 U 5 N;
     ITEM STATBIT B 3;            ITEM STB2 B 3;
     ITEM ANGLE S 4;              ITEM ANG2 S 4;
     END                          END
```

For each of the following, diagram the layout of the table. Assume that BITSINWORD is 32, M (medium packing) implies packing on the half word (bit 16), and that the compiler does not rearrange items within a table entry.

1. TABLE TRACK1 (1:100); (that is, as shown above)
2. TABLE TRACK1 (1:100) D;
3. TABLE TRACK2 (1:100) D;
4. TABLE TRACK1 (1:100) M;
5. TABLE TRACK2 (1:100) M;

ANSWERS

1\.

0 — 31
D(1)
S(1)
A(1)
⋮
D(100)
S(100)
A(100)

300 words

2\.

0 — 4	5 — 7	8 — 12	— 31
D (1)	S (1)	A (1)	
⋮			
D (100)	S (100)	A (100)	

100 words

ANSWERS

3\. 200 words

0 2	3 7	8 31
D2(1)		
S2 (1)	A2 (1)	

⋮

0 2	3 7	8 31
D2(100)		
S2 (100)	A2 (100)	

4\. 200 words

0 15	16 31
D(1)	S(1)
A(1)	

⋮

0 15	16 31
D(100)	S(100)
A(100)	

5\. 200 words

0 15	16 31
D2(1)	
S2(1)	A2(1)

⋮

0 15	16 31
D2(100)	
S2(100)	A2(100)

LIKE OPTION WITH PACKING

A table may have dense, medium or normal packing. When a table-type is declared using a like-option, packing does not apply to items specified within the like-option. For example, given the following declarations:

```
TYPE TEST TABLE D;
        BEGIN
        ITEM READ B 1;
        ITEM SET B 1;
        END
TYPE RETEST TABLE
        LIKE TEST;
        ITEM GO B 1;
```

Item GO, declared within the like-option, is not densely packed.

STRUCTURE

Table structure refers to the way in which whole entries of a table are laid out in memory. JOVIAL (J73) permits two fundamental types of structure, serial and parallel.

Serial Structure: The compiler lays out a serial table by taking the first word of the first entry, followed by the second word of the first entry, and so on. The entire first entry is allocated in this way, followed by the entire second entry, and so on.

A serial table may be structured as either an ordinary serial table, in which the compiler starts each entry in a new word, or a tight serial table, in which the compiler allocates as many entries as possible within a single word.

Parallel Structure: A table with PARALLEL structure may be specified only for a table in which none of the items of an entry occupy more than one word. A table is layed out in a parallel structure on a word-by-word basis.

The compiler lays out a parallel table by taking the first word (word 0) of the first entry followed by the first word of the second entry and so on to the first word of the last entry, then the second word (word 1) of the first entry, the second word of the second entry, and so on.

Examples

Consider the following declarations:

```
TABLE SBOX(1:2);                TABLE PBOX(1:2) PARALLEL;
        BEGIN                           BEGIN
        ITEM HEIGHT U;                  ITEM HEIGHT U;
        ITEM WIDTH U;                   ITEM WIDTH U;
        ITEM LENGTH U;                  ITEM LENGTH U;
        END                             END
```

The structure of these tables can be diagrammed as follows:

SBOX (serial)	**PBOX (parallel)**
HEIGHT(1)	**HEIGHT(1)**
WIDTH(1)	**HEIGHT(2)**
LENGTH(1)	**WIDTH(1)**
HEIGHT(2)	**WIDTH(2)**
WIDTH(2)	**LENGTH(1)**
LENGTH(2)	**LENGTH(2)**

The structure-spec is given in the table declaration following the parenthesized dimension-list, as follows:

```
TABLE  name  [ ( dimension-list ) ] [ structure-spec ]
        [ packing-spec ];
                entry-description
```

The square brackets indicate that the dimension-list, structure-spec, and packing-spec are optional. Although the dimension-list is optional in a table-declaration, a structure-spec is meaningful only when the table is dimensioned.

Structure-spec is one of the following:

PARALLEL

T [bit-per-entry]

The square brackets indicate that bits-per-entry is optional.

The letter T indicates tight serial structure. Bits-per-entry is an integer formula known at compile-time that gives the number of bits allocated for each entry. If bits-per-entry is not given, the compiler uses the minimum number of bits necessary to represent the entry for bits-per-entry. If no structure-spec is given, the compiler assumes that the table is an ordinary serial table.

Examples

1) Consider the following table declarations and layouts:

```
TABLE CLASS1 (1,1);
        BEGIN
        ITEM SUB1 U 5;
        ITEM LV1 U 3;
        ITEM HR1 U 6;
        END
```

BIT 0 — 31
SUB1 (0,0)
LV1 (0,0)
HR1 (0,0)
SUB1 (0,1)
LV1 (0,1)
HR1 (0,1)
SUB1 (1,0)
LV1 (1,0)
HR1 (1,0)
SUB1 (1,1)
LV1 (1,1)
HR1 (1,1)

```
2)      TABLE CLASS2 (1,1) PARALLEL;
            BEGIN
            ITEM SUB2 U 5;
            ITEM LV2 U 3;
            ITEM HR2 U 6;
            END
```

0 — 31
SUB2 (0,0)
SUB2 (0,1)
SUB2 (1,0)
SUB2 (1,1)
LV2 (0,0)
LV2 (0,1)
LV2 (1,0)
LV2 (1,1)
HR2 (0,0)
HR2 (0,1)
HR2 (1,0)
HR2 (1,1)

3) TABLE CLASS3 (1,1) T 16;

```
BEGIN
ITEM SUB3 U 5;
ITEM LV3 U 3;
ITEM HR3 U 6;
END
```

0	13	16	29 31
entry (0,0) **SUB3, LV3, HR3**		**entry (0,1)** **SUB3, LV3, HR3**	
entry (1,0) **SUB3, LV3, HR3**		**entry (1,1)** **SUB3, LV3, HR3**	

4) *Given the following declaration:*

```
TABLE CLASS(1:45) T;
    BEGIN
    ITEM SUBJ U 5;
    ITEM LV U 3;
    ITEM HOURS U 6;
    END
```

Default packing for a table with tight-structure is dense.

If BITSINWORD is 32, two entries can be packed per word, as follows:

```
                        Bit     0   4/5  7/8  13/14  18/19 21/22  27/    31
                               ----------------------------------------------
CLASS(1), CLASS(2)             | SUBJ LV HOURS SUBJ LV HOURS xxxx |
CLASS(3), CLASS(4)             | SUBJ LV HOURS SUBJ LV HOURS xxxx |
                               ----------------------------------------------
                                          ...
                               ----------------------------------------------
CLASS(45)                      | SUBJ LV HOURS xxxxxxxxxxxxxxxxx |
                               ----------------------------------------------
```

If BITSINWORD is 48, three entries can be packed per word, as follows:

```
Bit  0                13|14               27|28              41|       47
     ---------------------------------------------------------------------
     | SUBJ LV HOURS SUBJ LV HOURS SUBJ LV HOURS xxxxxx |
     | SUBJ LV HOURS SUBJ LV HOURS SUBJ LV HOURS xxxxxx |
     ---------------------------------------------------------------------
                 ...
     ---------------------------------------------------------------------
     | SUBJ LV HOURS SUBJ LV HOURS SUBJ LV HOURS xxxxxx |
     ---------------------------------------------------------------------
```

NOTE: When an item is declared outside of a packed table its implemented precision, the number of bits it is actually allocated, may be more than its declared precision, the number of bits declared for its size. When an item is declared in a packed table, the implemented precision is the same as the declared precision. An assignment to an item in a packed table may result in a loss of significant digits.

PACKING AND STRUCTURE -- EXERCISES

Indicate what is "wrong" with the following declarations:

1.
```
TABLE BOX PARALLEL;
      BEGIN
      ITEM HEIGHT U;
      ITEM WIDTH U;
      ITEM LENGTH U;
      END
```

2.
```
TABLE DATA(100) T;
      ITEM DATAPOINT U;
```

3.
```
TABLE RETURNS(99) D;
      BEGIN
      ITEM DATE U;
      ITEM PRIORITY F N;
      END
```

Assuming BITSINWORD is 32, indicate how many words the following tables occupy.

4.
```
TABLE DATA(1:10) D;
      BEGIN
      ITEM POINT U 5;
      ITEM XCOORD U 5;
      ITEM YCOORD U 5;
      END
```

PACKING AND STRUCTURE -- EXERCISES

5.
```
TABLE DATA(1:10) T 16;
    BEGIN
    ITEM POINT U 5;
    ITEM XCOORD U 5;
    ITEM Y COORD U 5;
    END
```

6.
```
TABLE DATA (1:10) D;
    BEGIN
    ITEM POINT U 5 N;
    ITEM XCOORD U 5;
    ITEM YCOORD U 5;
    END
```

ANSWERS

Indicate what is "wrong" with the following declarations:

1. TABLE BOX PARALLEL; — PARALLEL in undimensioned has no effect.

```
BEGIN
ITEM HEIGHT U;
ITEM WIDTH U;
ITEM LENGTH U;
END
```

2. TABLE DATA(100) T; — Tight has no effect when entries take more than halfword.

```
ITEM DATAPOINT U;
```

3. TABLE RETURNS(99) D; — Dense has no effect when no packing can be done.

```
BEGIN
ITEM DATE U;
ITEM PRIORITY F N;
END
```

Assuming that BITSINWORD is 32, indicate how many words the following tables occupy.

4. TABLE DATA(1:10) D; — 10 words

```
BEGIN
ITEM POINT U 5;
ITEM XCOORD U 5;
ITEM YCOORD U 5;
END
```

ANSWERS

5. TABLE DATA(1:10) T 16; 5 words

```
TABLE DATA(1:10) T 16;
    BEGIN
    ITEM POINT U 5;
    ITEM XCOORD U 5;
    ITEM Y COORD U 5;
    END
```

6. TABLE DATA (1:10) D; 20 words

```
TABLE DATA (1:10) D;
    BEGIN
    ITEM POINT U 5 N;
    ITEM XCOORD U 5;
    ITEM YCOORD U 5;
    END
```

LIKE-OPTION WITH STRUCTURE

A table may have serial (by default), tight or parallel structure. When a table type is declared using a like-option, the following constraints hold:

1) Bits-per-entry, in tight tables, also includes entries declared in the like-option.

2) A table obtained from like-option must have the same structure as the table-type using it.

Example

```
TYPE TEST TABLE T 16;

        BEGIN

        ITEM READY B 1;

        ITEM SET B 1;

        END

TYPE RETEST TABLE T 16;

        LIKE TEST;

                ITEM GO B 1;
```

THE JOVIAL (J73) WORKBOOK

VOLUME 12

APPLICATIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 12 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

Dynamic storage allocation is the application discussed in the following pages. The structure and content of this workbook follow almost exactly that of the videotape. Examples of J73 code given in the videotape are repeated, along with explanations.

Section 1 illustrates three ways to create a linked list (two of which are discussed in the tape). Section 2 addresses the creation, allocation and deallocation of storage. It is suggested that the student follow along in the workbook while viewing the videotape.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction-a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

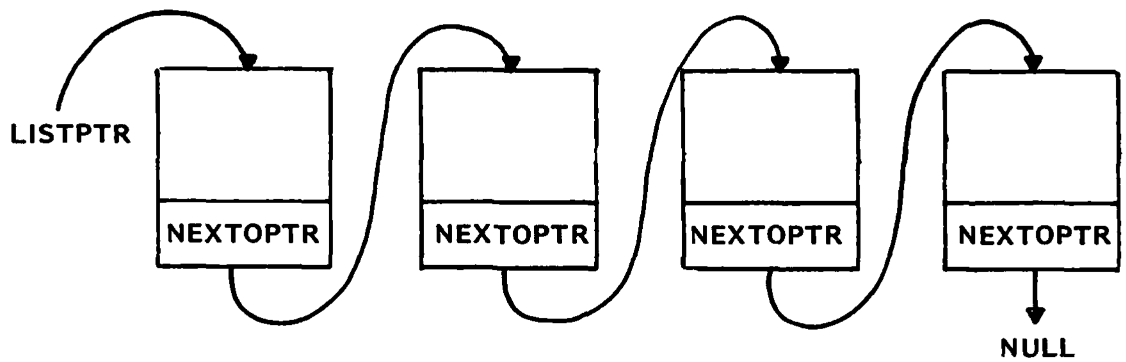
SECTION 1

LINKED LISTS

SOFTECH

LINKED LISTS

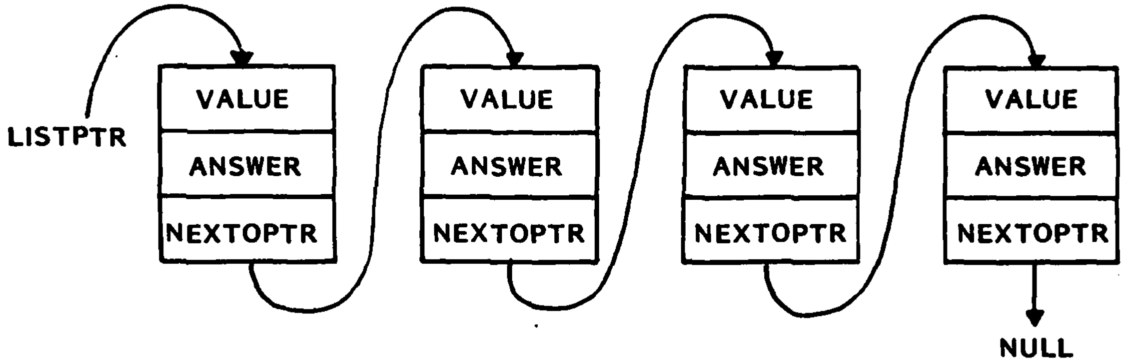
A programmer may use dynamic storage when a program requires a constantly changing data configuration. One example of this kind of data structure is a linked list. A linked list could be diagrammed as follows:



This diagram shows four *data objects linked together* by an item, NEXTOPTR. The NEXTOPTR at the end of the list points to NULL, and LISTPTR points to the head of the list.

The following declarations give information about the representation of the elements in the linked list:

```
TYPE LISTYPE TABLE;
       BEGIN
       ITEM VALUE U;
       ITEM ANSWER B 4;

ITEM NEXTOPTR P LISTYPE;
       END

ITEM LISTPTR P LISTYPE;
ITEM TEMPTR P LISTYPE;
```



This list was created such that the value of LISTPTR is the address of the first element in the list, so the dereference @LISTPTR is used to reference items in that first element. For example, the first VALUE is referenced as follows:

    VALUE @ LISTPTR

Similarly, a reference to the item answer in the first element would look like this --

    ANSWER @ LISTPTR

The value of the first NEXTOPTR is the address of the second element in the list, so the dereference @(NEXTOPTR @ LISTPTR) is used to reference items in the second element. For example, the second VALUE is referenced by:

    VALUE @ (NEXTOPTR @ LISTPTR)

To reference items in the third element, an item name followed by three dereferences would be used --

    ANSWER @ (NEXTPTR @(NEXTPTR @ LISTPTR))

A way of simplifying this process somewhat would be to assign the address of the third element of the list to a temporary pointer as follows:

    TEMPTR = NEXTOPTR @ (NEXTOPTR @ LISTPTR);

That pointer may then be used in a data reference --

VALUE @ TEMPTR

CREATING LINKED LISTS

Linked lists don't "just happen;" VALUE, ANSWER, and NEXTOPTR are not just randomly grouped together; the NEXTOPTRs do not automatically point to another element. This data structure must be created by the programmer.

First, the programmer must write storage management routines to set up and maintain a heap of free storage, to allocate that storage, and to deallocate that storage and return it to the heap. (The heap is a large area of storage that may be supplied by an implementation or managed by the programmer.)

Section 2 defines algorithms suitable for this purpose, but for now assume that the following subroutines are available:

- MAKE'HEAP'LISTYPE -- to create a heap of storage
- NEW'LISTYPE -- to allocate storage from the heap
- FREE'LISTYPE -- to free allocated storage and return it to the heap

There are several ways to create the linked list pictured above. To begin with, the first element of the list is allocated by calling the NEW'LISTYPE function, which returns a pointer whose value is the address of a new piece of storage of type LISTYPE. The address of this piece of storage is assigned to be the value of LISTPTR, which is used to indicate the head of the list --

LISTPTR = NEW'LISTYPE;

The second element of the list is added by calling the NEW'LISTYPE function and assigning the returned pointer to new storage to be the value of the NEXTOPTR at the element pointed to by LISTPTR --

NEXTOPTR @ LISTPTR = NEW'LISTYPE;

The third element is added in the same manner --

```
NEXTOPTR @ (NEXTOPTR @ LISTPTR) = NEW'LISTYPE;
```

as is the fourth element --

```
NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @ LISTPTR)) = NEW'LISTYPE;
```

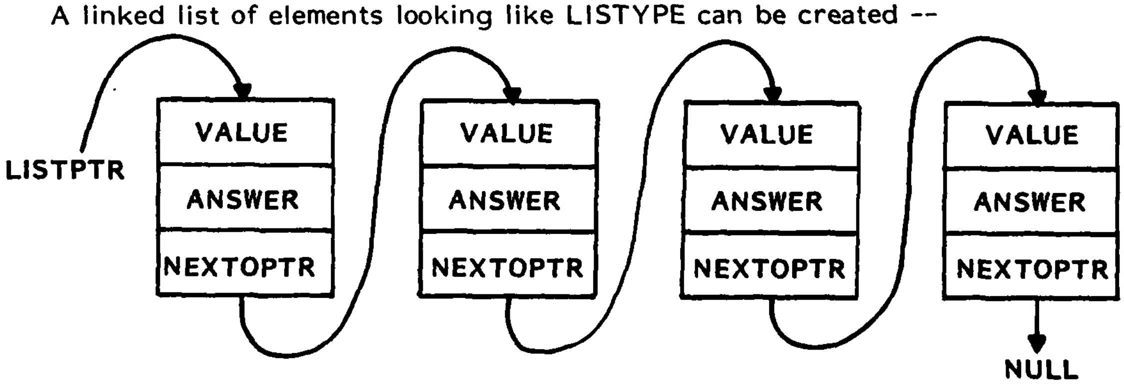
The value of the last NEXTOPTR is set to NULL to indicate the end of the list --

```
NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @
     LISTPTR))) = NULL;
```

Thus, using the following code:

```
TYPE LISTYPE TABLE;
     BEGIN
     ITEM VALUE U;
     ITEM ANSWER B 4;
     ITEM NEXTOPTR P LISTYPE;
     END

ITEM LISTPTR P LISTYPE;


LISTPTR = NEW'LISTYPE;                                (one link)

NEXTOPTR @ LISTPTR = NEW'LISTYPE;                     (two links)

NEXTOPTR @ (NEXTOPTR @ LISTPTR) = NEW'LISTYPE;        (three links)

NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @ LISTPTR)) =        (four links)
NEW'LISTYPE;

NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @ (NEXTOPTR @        (last pointer
LISTPTR))) = NULL;                                    set to NULL)
```

A linked list of elements looking like LISTYPE can be created --



By introducing another pointer item, used to indicate the current end of the list, and by using a for-loop, the previous code becomes simplified --

```
TYPE LISTYPE TABLE;
       BEGIN
       ITEM VALUE U;
       ITEM ANSWER B 4;
       ITEM NEXTOPTR P LISTYPE;
       END
ITEM LISTPTR P LISTYPE;
ITEM TAILPTR P LISTYPE;

LISTPTR = NEW'LISTYPE;                          (one link)
TAILPTR = LISTPTR;                              (three times do:)
FOR I : 1 BY 1 WHILE I <= 3;
       BEGIN
       NEXTOPTR @ TAILPTR = NEW'LISTYPE;        (add new link to end)
       TAILPTR = NEXTOPTR @ TAILPTR;            (move temporary ptr)
       END
NEXTOPTR @ TAILPTR = NULL;                      (set ptr at end to
                                                 NULL)
```

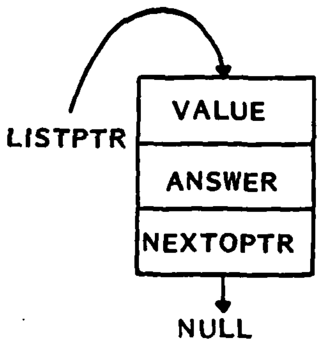
The first element in the list is allocated by calling NEW'LISTYPE, and its address is assigned to LISTPTR. Since it also represents the current final element in the list, TAILPTR also points to it.

The remaining three elements are added in the for-loop. First, a new piece of storage is obtained, and its address becomes the value of NEXTOPTR at the current end of the list. Then, TAILPTR is set to indicate the new end of the list. This process is repeated two more times, until all four elements are allocated and linked together. Finally, the NEXTOPTR at the current end of the list is set to NULL to indicate the end of the list.

The third method of creating a linked list, alluded to in the tape, involves the inverse process -- i.e., creating the linked list from NULL. The following code may be used to accomplish this --

```
TYPE LISTYPE TABLE;
        BEGIN
        ITEM VALUE U;
        ITEM ANSWER B 4;
        ITEM NEXTOPTR P LISTYPE;
        END
ITEM LISTPTR P LISTYPE;
ITEM TEMPTR P LISTYPE;

LISTPTR = NEW'LISTYPE;
NEXTOPTR @ LISTPTR = NULL;
FOR I : 1 BY 1 WHILE I <= 3;
        BEGIN
        TEMPTR = NEW'LISTYPE;
        NEXTOPTR @ TEMPTR = LISTPTR;
        LISTPTR = TEMPTR;
        END
```

This program piece creates a linked list from the bottom up. The NEXTOPTR of the first element is set to NULL and new elements are created and added to the list between the LISTPTR and previous elements.



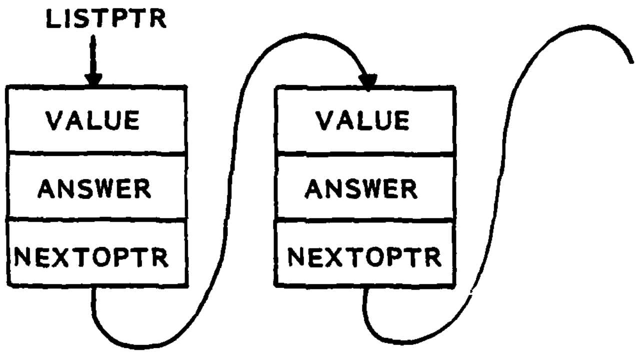
REMOVING ELEMENTS FROM A LINKED LIST

An element does not "just disappear" from a linked list. It must be deallocated and the freed storage area returned to the built-in or programmer-defined heap storage. This is done by using the FREE'LISTYPE procedure. The FREE'LISTYPE procedure is called with a pointer-formula as input.

Using the linked list structure created above the following example removes the topmost element.

Example

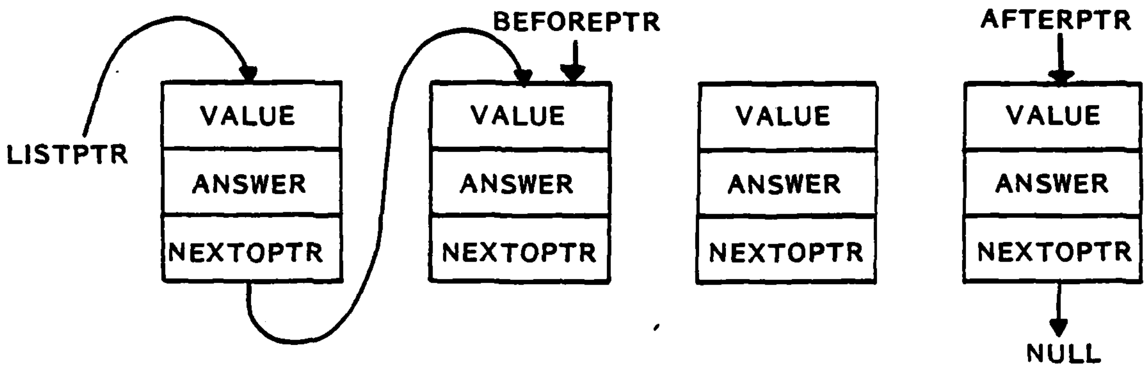
```
SAVEPTR = NEXTOPTR @ LISTPTR;

FREE'LISTYPE(LISTPTR);

LISTPTR = SAVEPTR;
```



First the address of the second element in the list is saved. Then, the FREE'LISTYPE procedure is called with LISTPTR so the storage for the first element is returned to the heap. Finally, LISTPTR is reassigned to indicate the new head of the list.

Freeing an element from the middle of the list is a bit more complicated. The following program fragment frees the third element of the list and returns it to the heap:

```
TYPE LISTYPE TABLE;
        BEGIN
        ITEM VALUE U;
        ITEM ANSWER B 4;
        ITEM NEXTOPTR P LISTYPE;
        END

ITEM LISTPTR P LISTYPE;
ITEM AFTERPTR P LISTYPE;
ITEM BEFORE PTR P LISTYPE;

BEFOREPTR = NEXTOPTR @ LISTPTR;
AFTER PTR - NEXTOPTR @ (NEXTOPTR @ BEFOREPTR);
FREE'LISTYPE (NEXTOPTR @ BEFOREPTR);
NEXTOPTR @ BEFOREPTR = AFTERPTR;
```



BEFOREPTR is used to save the parts of the list before the elements to be freed; AFTERPTR is used to save the rest. FREE'LISTYPE is called to return the selected element to the heap, and the remaining parts of the list are linked.

SORTING A LINKED LIST

In some applications, linked lists may need to be sorted for quicker access of certain elements. One method of sorting a list involves reassigning pointers to change the order in which the list is traversed; this effectively reorders the elements. In the following example, the NEXTOPTRs will be reassigned such that the third element will be traversed before the second.

```
TYPE LISTYPE TABLE;
       BEGIN
       ITEM VALUE U;
       ITEM ANSWER B 4;
       ITEM NEXTOPTR P LISTYPE;
       END

ITEM LISTPTR P LISTYPE;
ITEM THIRDPTR P LISTYPE;
ITEM SECONDPTR P LISTYPE;
ITEM SAVEPTR P LISTYPE;

THIRDPTR = NEXTOPTR @ LISTPTR;
SECONDPTR = NEXTOPTR @ THIRDPTR;
SAVEPTR = NEXTOPTR @ SECONDPTR;
NEXTOPTR @ LISTPTR = SECONDPTR:
NEXTOPTR @ SECONDPTR = THIRDPTR;
NEXTOPTR @ THIRDPTR = SAVEPTR;
```

THIRDPTR is set to indicate the element that will be traversed third; SECONDPTR is set to indicate the element that will be traversed second; and SAVEPTR is set to save the end of the list. The NEXTOPTRs are reassigned as follows:

a. the value of the first NEXTOPTR is assigned to be the address of the element pointed to by SECONDPTR.

b. the value of the now second NEXTOPTR is assigned to be the address of the element pointed to by THIRDPTR.

c. the now third NEXTOPTR is linked to the saved portion of the list.

SECTION 2

SAMPLE STORAGE MANAGEMENT ROUTINES

SAMPLE STORAGE MANAGEMENT ROUTINES

This section looks at the storage management routines shown in Tape 12, which were designed to be used with a list consisting of elements of type LISTYPE.

Since the storage management routines were designed by one programmer to be used easily by another, two compool-modules were written.

```
START
COMPOOL TYPES;
        TYPE LISTYPE TABLE;
        BEGIN
        ITEM VALUE U;
        ITEM ANSWER B 4;
        ITEM NEXTOPTR P LISTYPE;
        END

        TYPE HEAP'LISTYPE TABLE
                (1:100) LISTYPE;
TERM

START
!COMPOOL ('TYPES');
COMPOOL REFS;
        REF PROC MAKE'HEAP'LISTYPE;
                BEGIN
                END
        REF PROC NEW'LISTYPE P LISTYPE;
                BEGIN
                END
        REF PROC FREE'LISTYPE (IN'PTR);
                ITEM IN'PTR P LISTYPE;
        REF PROC ERROR (IN'MSG);
                ITEM IN'MSG C 20;
TERM
```

The first compool-module contains type-declarations -- LISTYPE, which is used as the template for elements in the linked list, and HEAP' LISTYPE, which collects one hundred elements of type LISTYPE. HEAP'LISTYPE is used as the template for the heap storage, so the largest linked list that may be created with these storage management routines could only be one hundred elements long.

The second compool-module contains the REF-procs for the storage management routines and an error routine.

Compool TYPES is imported so the compiler has access to type LISTYPE which is used in the subroutine-declarations.

Once again -- by placing the REF-procs in a compool-module and later importing that compool-module into the module containing the DEF-procs, the compiler is able to check that the correct calling sequence is given in the REF-procs and that the subroutines as defined are called in the main-program-module.

The following procedure module imports both compools, one to make the types available and the other to do the DEF/REF checking. Two local, static data objects are declared. HEAPTR is used to indicate the head of the heap storage; HEAPTAB is the table that is used to create the heap storage.

```
START
!COMPOOL ('REFS');
!COMPOOL ('TYPES');
ITEM HEAPTR P LISTYPE;
TABLE HEAPTAB HEAP'LISTYPE;
DEF PROC MAKE'HEAP'LISTYPE;
      BEGIN
      ITEM TEMPTR P LISTYPE;
      ITEM UPPER U;
      ITEM LOWER U;
      UPPER = UBOUND (HEAPTAB, 0);
      LOWER = LBOUND (HEAPTAB, 0);
      HEAPTR = LOC (HEAPTAB (UPPER));
      NEXTOPTR @ HEAPTR = NULL;
      FOR I : UPPER BY -1 WHILE I <= LOWER:
            BEGIN
            TEMPTR = LOC (HEAPTAB)(I-1));
            NEXTOPTR @ TEMPTR = HEAPTR;
            HEAPTR = TEMPTR;
            END
      END
```

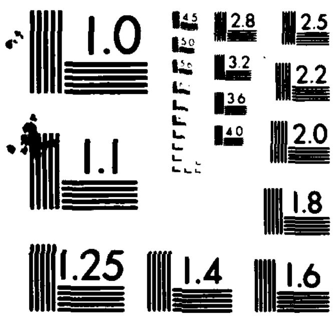
```
DEF PROC NEW'LISTYPE P LISTYPE;
      BEGIN
      ITEM TEMPTR P LISTYPE;
      IF HEAPTR = NULL;
            ERROR (NEW'LISTYPE);
      ELSE
            BEGIN
            TEMPTR = HEAPTR;
            HEAPTR = NEXTOPTR @ TEMPTR;
            NEXTOPTR @ TEMPTR = NULL;
            NEW'LISTYPE = TEMPTR;
            END
      END

DEF PROC ERROR (IN'MSG);
      BEGIN
      ITEM IN'MSG C 20;
      !TRACE IN'MSG;
      IN'MSG = IN'MSG;
      STOP;
      END


DEF PROC FREE'LISTYPE (IN'PTR);
      BEGIN
      ITEM IN'PTR P LISTYPE;
      NEXTOPTR @ IN'PTR = HEAPTR;
      HEAPTR = IN'PTR;
      END
TERM
```

The first subroutine-definition creates the heap storage. It must be called once and only once by the program using the storage management routines to initialize the heap storage.

MAKE'HEAP'LISTYPE takes 100 unliked elements of type LISTYPE (that is what HEAP'LISTYPE described) and links them together, beginning with the largest subscripted element being the end of the list, then adding new elements to the head of the list.

AD-A108 528 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME 10. DIRECTIVES.(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-10 NL

END
DATE
FILMED
1 82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

When MAKE'HEAP'LISTYPE is finished executing, HEAPTR is pointing to the current head of HEAPTAB, the lowest subscripted element.

NEW'LISTYPE is defined to first check if there is available heap storage by checking the value of HEAPTR.

If HEAPTR is equal to NULL, no storage is available, and the name NEW'LISTYPE is passed to an error routine.

This error routine is defined to output the name of the routine with the error, in this case, NEW'LISTYPE, and then to stop program execution.

If storage is available, a temporary pointer indicates the storage to be allocated, HEAPTR is moved to what will be the head of the list after the element is allocated, the element to be allocated is unhooked from the heap, and allocated as the return-value of NEW'LISTYPE. FREE'LISTYPE is defined to assign the value of the NEXTOPTR of the element being returned to the heap to the head of heap and then HEAPTR is adjusted to indicate the new head of the heap storage list.

Once these storage management routines have been compiled, any programmer may use them in a program by importing the compool-modules with the type-declarations and the REF-procs.

The heap is initialized by calling MAKE'HEAP'LISTYPE once.

The NEW'LISTYPE function is called when the linked list is growing; the FREE'LISTYPE procedure is called when elements are to be returned to the heap.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DAT

FILM